7-2020

Python Implementation of Batch Least-Squares Filter for Satellite Orbit Determination

Austin Ogle

Python Implementation of Batch Least-Squares Filter for Satellite Orbit Determination

Austin Ogle

Thesis submitted to the Faculty of the
Embry-Riddle Aeronautical University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Engineering Physics

Ed Mierkiewizc, Chair
Stephen Gillam, Advisor and co-chair
Chad Brodel
Muhammad Farooq
Sirani M. Perera

July 22, 2020
Daytona Beach

Keywords: Orbit Determination, Space Debris, Least-Square Filter, Satellite Tracking

PYTHON IMPLIMENTATION OF BATCH LEAST SQUARES FILTER FOR ORBIT DETERMINATION

By

Austin J. Ogle

This thesis was prepared under the direction of the candidate's thesis committee Chair: Dr. Edwin Mierkiewicz, M.S. Program Coordinator in Engineering Physics & Associate Professor of Physics and Adviser and Co-chair: Dr. Stephen Gillam Assistant Professor of Physics and Astronomy, Daytona Beach campus, and has been approved by the thesis committee. It was submitted to the Department of Physical Sciences in partial fulfillment of the requirements of the degree of
Master of Sciences in Engineering Physics.

THESIS COMMITTEE:

Edwin Mierkiewicz
Digitally signed by Edwin Mierkiewicz
DN: cn=Edwin Mierkiewicz, o, ou, email=mierklee@erau.edu, c=US
Date: 2020.07.30 07:45:29 -06'00'

Dr. Edwin Mierkiewicz
Committee Chair

Steve Gillam

Dr. Stephen Gillam
Committee Co-Chair and Adviser

Dr. Sirani M. Perera
Committee Member

Omer Farooq

Dr. Muhammad Farooq
Committee Member

Dr. Chad Brodel
Committee Member

Terry D. Oswalt

Dr. Terry Oswalt
Department Chair, Physical Sciences

Karen F. Gains

Dr. Karen Gains
Dean, College of Arts and Sciences

Christopher Grant
Digitally signed by Christopher Grant
Date: 2020.08.12 15:42:21 -04'00'

Dr. Christopher Grant
Associate Provost of Academic Support

ABSTRACT

Accurate orbit determination techniques are fundamental to the maintenance and execution of any ongoing space-based mission. This project serves as a guide and demonstration of a batch sequential least-squares filter for Earth-orbiting satellites using exclusively open-source technologies. The target audience for this project was an academic institution aiming to keep track of an irregularly documented satellite. The observation function mimics a telescope, accepting right ascension and declination as measured values.

State propagation was handled using the Poliastro library. This package boasts FORTRAN-level speed by utilizing the DOPRI8 integrator, explicitly calling FORTRAN code. Matrix inversion was solved using the SciPy banded_solver function, a wrapper for the LAPACK dgbsv function, also written in FORTRAN. Frame conversions between ITRS (ECEF), GCRS (ECI), and ICRS (J2000) were handled using Astropy.

A suite of tests with a range of noise were run to verify appropriate convergence of algorithm. In each case, the algorithm converged as expected with reasonable variances that changed in an anticipated fashion. These tests demonstrated that it is possible to achieve sub km accuracy for LEO satellites with 10 observations given 1 arcminute uncertainty and noise.

Despite the interface requiring manual, the backend has been optimized to save memory supporting large batches of observations. As a result, the project detailed in this report requires little adaptation to support a much larger scale use such as tracking orbital debris. Any such changes are outlined in the designing a system subsection 3.3.1 or future expansion chapter 8.

A GUI was assembled to support users with a limited coding background using Kivy.

Acknowledgments

I would like to thank my friends and family who have supported me thus far. This wouldn't have been possible without you all.

"[I]'ll pass the test, and finish the quest"

-*Unknown*

Contents

List of Figures

List of Tables

List of Symbols

J - Cost function
$\vec{\rho}$ - Residual observational vector
$\boldsymbol{\rho}$ - Residual matrix
ρ - Residual scalar
$\vec{y}$ - Observation vector
$\boldsymbol{Y}(\cdot)$ - Observation function
$\vec{x}$ - State vector
$\vec{\xi}$ - Calculated residual observational vector
$\boldsymbol{A}$ - Partial Derivative matrix or normal matrix in normal equation
$\vec{b}$ - Right hand side vector in normal equation
$\boldsymbol{P}$ - Covariance matrix
$\boldsymbol{W}$ - Weighted matrix
$\boldsymbol{\Lambda}$ - Information Matrix
$\Delta\bar{x}^{apr}$ - A priori update to the state vector
$\boldsymbol{P}^{apr}$ - A priori covariance matrix
a - Semi-major axis
e - Eccentricity
i - Inclination
Ω - Right ascension of the ascending node
ω - Arguement of periapsis
θ - True anomaly
$\vec{r}$ - Position
$\vec{v}$ - Velocity
$\vec{a}$ - Acceleration
$\boldsymbol{\Phi}^t_{t_0}$ - State transition matrix
$\mathbb{M}$ - Generic matrix
$\boldsymbol{\Pi}$ - Generic matrix product
$\boldsymbol{ab}$ - Diagonalized normal form
RMS - Root mean squared scalar
σ - Standard deviation
μ - Gravitational constant
x - x-component of position
y - y-component of position
z - z-component of position
r - norm of position
R - Radius of the Earth
ρ - atmospheric density
C_d - Coefficient of drag
A - Ram surface area
m - Mass
M - Mean anomaly
E - Eccentric anomaly
α - Right ascension
δ - Declination
$r\vec{r}$ - Vector from observer to target

Chapter 1

Background

Not to be confused with initial orbit determination, orbit determination is the process of maintaining an accurate description of a desired object's orbit. For every space-based mission, the location of the satellite is fundamental. At a minimum, communication systems require accurate pointing and therein situational awareness. To maintain that awareness, there are two related but distinct techniques: Least-Squares (LSQ) Filter and the Kalman Filter. Both rely on similar principles, however, are fundamentally different in implementation. Depending upon the nature of the mission, one or both are reasonable to implement. Before we delve further, I would like to identify a misnomer and the most fundamental difference between these two techniques. The Least-Squares Filter is not a filter, but rather a smoothing process [15]. It does not bring the object's location and uncertainty forward i n time, unlike a filter would. Instead, the LSQ approach converges to a more accurate estimate of the orbit at an initial epoch with a described uncertainty. These can be propagated forward in time but is not done inherently in the LSQ filter.

The choice between LSQ and Kalman filters is often decided by the mission. While both can provide the same level of accuracy [9] [4], Kalman filters are preferred for real-time problems while LSQ filters are often used back-ward looking on existing data. Hidden in the real-time assumption is the requirement for constant support or at least computing power. For missions where the users are unable to spare the memory/computational time or lack the regular real-time measurements, a sequential batch LSQ filter would be applicable. Batch refers to handling a clump of data at a time, while sequential implies remembering the smoothing effects of previous batches. Without the word batch, a sequential LSQ filter is merely a Kalman filter.

When considering real-time missions that would benefit from a LSQ filter, two circumstances come to mind. Both stem from a desire for a real-time orbit definition paired with infrequent observations. The first includes a national agency tasked with tracking pieces of orbital debris for collision avoidance purposes. This entails describing the trajectory of each object with uncertanties. Maintaining real-time updates with a Kalman filter would require continual

updates to the covariance without any new observations. With a LSQ filter, this significant computational cost would be possible on a need-be basis. The second mission would be tracking of a satellite for a small company or organization that is not large enough to be picked up by open-source tle generators. If the parent organization can observe their own object, applying a LSQ filter could prove invaluable in refining a trajectory.

Orbit determination is just one of the many problems that can be solved with a LSQ Filter or a Kalman Filter. Both techniques are mathematical algorithms that are not tied directly to the physical world. A LSQ filter directly relies upon an observation function, which relies upon a state propagator. The observation function can mimic any sensor such as radar, gps, or telescope. The state propagator encompasses all of the physics of the environment, potentially astrodynamics. In the case of underwater navigation, a state propagator would look much different. When applying a LSQ filter another problem, the observation function and state propagator would need to be swapped out, but the LSQ logic could remain in its entirety.

Chapter 2

Introduction

The primary goal of this project was to implement a least-squares filter for orbit determination to be used by an academic institution. Universities have been known to launch small satellites or cubesats which may not receive updated TLEs from traditional open-sources, this tool would serve as a mean to provide their own orbit estimation. Implicitly, the interface should be user-friendly, such that an undergraduate can use it. The only skill/experience required would be related to operating the telescope itself. Notably, this also assumes the satellite will be observed exclusively via telescope. As a result, the following restrictions were set upon the project:

1. No licenses required
2. No coding experience required; a GUI is included
3. Capable of running on a windows environment

The no license restriction placed significant limitations on design as it ruled out a lot of the technologies the author had experience with. Notably, MATLAB and STK were no longer viable technologies to be included as both require particularly expensive licenses. While a prototype was built in MATLAB, the final product is exclusively written in Python.

The software development phase was agile-like. No scrum occurred as there was only one team member. Sprints lasted a week. I met weekly with Dr. Gillam, my advisor, who served as the product-owner. Each meeting, we went over what tickets were accomplished and prioritized and selected tickets for the upcoming sprint. This meeting also allowed Dr. Gillam to see what was accomplished. The git repository mirrored this philosophy. There were three primary branches, active, current_sprint, and master. Each ticket was accomplished on active, then merged to current_sprint. Current_sprint was only pushed to master during the sprint reviews after Dr. Gillam had the opportunity to see all changes. This allowed for a very clear progress report each week.

The project was written with Test Driven Development (TTD) in mind. Embracing this philosophy involves writing the test for the production code first, then writing the production code. While this adds a lot of overhead to the development process, it ensures the code was well-thought out and verified to work. Additionally, it also allowed for a parallel set of code that could be run to ensure that everything was working as intended. At one point in time, days were spent trying to identify an issue that could have been easily found if unit tests were executed after making a perceived improvement to the code.

2.1 Technologies Used

2.1.1 Propagator

From the outset, it was clear that state propagation was going to be a critical component of the project. Due to the desire for a high-accuracy model, including perturbations was going to be required. Consequently, Cowell's method was the most practical approach. Cowell's method merely involves integrating force over time. This allows us to include the effects of perturbations in their most well-known form, the force they create and not some strange geospatial impact on a trajectory, as Enke's method requires [1]. In total, this requires accurate descriptions of the perturbing forces and a robust integrator. After STK, GMAT was considered for its high accuracy. However, it was not going to be possible to build a GUI in the GMAT scripting language. The next best alternative was Python with its many open-source libraries and vast capabilities.

After considering a number of libraries for orbit propagation, Poliastro was deemed to be the best fit. It featured a solid list of perturbing forces and used the Prince-Dormund 8th order SciPy integrator. All of this will be discussed further in depth in the Propagation chapter of this report, however, I would like to add that Poliastro featured everything I set out for. Most importantly, Poliastro was user-friendly and boast "FORTRAN-levels of speed" [12]. A notable alternative includes the SGP4 model which has a python wrapper/implementation. Compared to poliastro, I found this model to be very difficult to interact with and nearly impossible to read. Towards the end of my project, I learned that the SGP4 model was directly tied to Two-Line Elements (tle) and NASA/NORAD [14]. Utilizing Poliastro has the side effect of making it difficult for this project to work with TLEs, while working with SGP4 would have the opposite effect. It would be difficult to work with satellites without a TLE description.

2.1.2 Unit Testing

In the production code of this project, every line of code is unit tested. To accomplish this, a number of packages were used together, including: pytest, mockito, and pytest-cov. Pytest

and Mockito were both chosen as the author had previous experience with both. Pytest-cov was used to ensure that full coverage was met. Mockito was identified not merely due to previous experience, but also because it seemed to offer something unique. In the past, the author worked with Mockito in Java and after being unable to find desired functionality in mainstream python unittesting packages, the author turned to Mockito.

The foundation of TDD is unit tests. Unit tests are nearly self-explanatory, they test the lowest meaningful unit - a function. Depending upon the number of if-statements or logic paths, a function may require more than one test. The goal of a unit test is to ensure the function is working exactly as intended. In order to do this, a function must be tested independent of all the other functions it may or may not interact with. Should the function being tested call other functions, their responses are mocked to ensure no cross-contamination of errors.

Mockito provides the ability to stub functions and provide custom results based upon their input. From my research, the main unit test packages allowed stubbing but with the exception of Mockito, none verified the inputs matched what was expected. Without this capability, unit tests would only be able to verify the section of the function since it was called externally and not the full function. A great example of this sort of stubbing would be the unit tests for the dx_dstate function in test_core.py file, see appendix, as it requires 24 total stubbed responses. Additionally, Mockito provided a very clear function for stubbing results and was incredibly easy to read. When dealing with Mockito it is important to note that it cannot verify if two NumPy arrays are the same. To accomplish this, the xcompare function was written and is included in the test/ python package. It's implementation in a test can also bee seen the in the aforementioned unit test dx_dstate in test_core.py.

2.2 Structure of the Report

Much like the code itself, this project is conceptually modularized. As previously mentioned, the least squares filter is merely a technique and can be applied to a great number of problems. As it is the core of the project, it will be discussed first in chapter 3. In chapter 4, we will discuss the propagation method chosen. This chapter encompasses all of the astrodynamics within the project. The observation function will follow chapter 5. This serves as the bridge between astrodynamics and the least squares filter. In chapter 6, we will discuss the GUI. Following, we will discuss testing scenarios in chapter 7 and end with future improvements in chapter 8.

Chapter 3

Least Squares Filter

3.1 Theory

A least squares filter is a smoothing process that aims to fit data by minimizing a cost function. There is no objectively true cost function. At first, we will explore the most basic cost function, one that measured the distance of the given points from the fit. Later on, we will expand the cost function to take into account additional information. x_0 is the initial estimate of the satellite state at the epochs of observations, indicated by the summation over l, the number of observations.

$$J(\vec{x}) = \sum_{l} \vec{\rho}^{T} \vec{\rho} = \sum_{l} (\vec{y}_{obs} - Y(\vec{x}_0))^T (\vec{y}_{obs} - Y(\vec{x}_0)) \tag{3.1}$$

$\vec{\rho}$ is the residual between the observations of the state of the satellite and the predicted observation values of the state. This vector is of length n, dependent upon the observation function and summed across l observations. In our case, the observation function mimics a telescope and consists of two angles. If the observation mimicked radar systems, this vector would consist of three components, range and two angles. Y is the observation function and converts a state into observable values. It can be approximated using a Taylor expansion around a reference trajectory, the ideal solution.

$$\vec{\rho} = \vec{y}_{obs} - Y(\vec{x}_0) = \vec{y}_{obs} - Y(\vec{x}_0^{ref}) - \frac{\partial Y}{\partial \vec{x}}(\vec{x}_0^{ref} - \vec{x}_0) - \frac{\partial^2 Y}{\partial \vec{x}^2}(\vec{x}_0^{ref} - \vec{x}_0)^2 ...$$

This expression can be simplified using the following substitutions.

$$\vec{\xi} = \vec{y}_{obs} - Y(\vec{x}_0^{ref})$$

$$\vec{x} = \vec{x}_0^{ref} - \vec{x}_0$$

This gives the following representation

$$\vec{\rho} = \vec{\xi} - \frac{\partial \vec{\xi}}{\partial \vec{x}}\vec{x} - \frac{\partial^2 \vec{\xi}}{\partial \vec{x}^2}(\vec{x})^2 ... \tag{3.2}$$

$\frac{\partial \vec{\xi}}{\partial \vec{x}}$ is a m x n matrix while $\frac{\partial^2 \vec{\xi}}{\partial \vec{x}^2}$ has the shape of m x n x n, where m is the size of the state vector $\vec{x}$. If $\vec{x}$ is small, then $\frac{\partial^2 \vec{\xi}}{\partial \vec{x}^2}$ provides little value despite being more expensive to compute. Similarly, going forward with further derivatives is even more costly with limited impact on the ability to converge. This requires our initial guess to be close to the solution. From now on, we will make the substitution $\boldsymbol{A} \equiv \frac{\partial \vec{\xi}}{\partial \vec{x}}$, which gives

$$\vec{\rho} = \vec{\xi} - \boldsymbol{A}\vec{x}$$

Rewriting the cost function, we can see

$$J = (\vec{\xi} - \boldsymbol{A}\vec{x})^T(\vec{\xi} - \boldsymbol{A}\vec{x}) \tag{3.3}$$

The cost function J is dependent upon observed values, treated as constants, and $\vec{x}$, the estimated state. The minimum of the cost function occurs when $\frac{\partial J}{\partial \vec{x}} = \vec{0}$.

Using the relation,

$$\frac{\partial \vec{A}^T \vec{B}}{\partial \vec{X}} = \vec{B}^T \frac{\partial \vec{A}}{\partial \vec{X}} + \vec{A}^T \frac{\partial \vec{B}}{\partial \vec{X}}$$

$$\frac{\partial J}{\partial \vec{x}} = 0 = -2(\vec{\xi} - \boldsymbol{A}\vec{x})^T \boldsymbol{A}$$

This equation is often represented in a simpler form

$$(\boldsymbol{A}^T \boldsymbol{A})\vec{x} = \boldsymbol{A}^T \vec{\xi} \tag{3.4}$$

To recap, $\vec{\xi}$ is the residual vector between truncated predicted and measured observational values. $\vec{x}$ is the update to the initial state. Lastly, $\boldsymbol{A}$ is $\frac{\partial \vec{\xi}}{\partial \vec{x}}$. This equation is solved with each iteration and the original $\vec{x}_0$ is updated appropriately.

It is relevant to point out that (3.4) closely resembles the normal equation. $\boldsymbol{A}\vec{x} = \vec{b}$. This implies that $(\boldsymbol{A}^T \boldsymbol{A})^{-1} = \boldsymbol{P}$ the covariance matrix. Isolating for $\vec{x}$, we see

$$\vec{x} = \boldsymbol{P}\boldsymbol{A}^T\vec{\xi} \tag{3.5}$$

3.1.1 Weighted Least Squares Filter

As it stands, a vital issue with the above algorithm is that all observations are weighed equally. In practice, it is often the case that a few highly accurate observations dominate a larger group of less accurate measurements. As such, it is important to add a weighted matrix that considers the relative "value" of each observation. Consequently, we define the $\boldsymbol{W}$ matrix as the inverse of the measurement error squared or

$$\boldsymbol{W} = \begin{bmatrix} \frac{1}{\sigma_1^2} & 0 & 0 & 0 \\ 0 & \frac{1}{\sigma_2^2} & 0 & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & \frac{1}{\sigma_n^2} \end{bmatrix}$$

In turn, the cost function is redefined below

$$J = \vec{\rho}^T\boldsymbol{W}\vec{\rho}$$

Ultimately, this has the impact of changing the normal equation derived above.

$$\vec{x} = (\boldsymbol{A}^T\boldsymbol{W}\boldsymbol{A})^{-1}\boldsymbol{A}^T\boldsymbol{W}\vec{\xi} \tag{3.6}$$

3.1.2 *A Priori* Information

The second significant improvement relies around including previous knowledge and alone adds the word "batch" to the title of this report. This technique is referred to as estimation with *a priori* information. Consider the circumstance where 1000 observations are incorporated in a single batch and a new, more accurate state is found. Shortly thereafter, 30 more measurements are taken and included. With the current theory all 1030 measures would then need to be iterated upon, which is particularly costly. By including *a priori*, we can run the 1000 measurements and 30 separately, but still benefit from their collective information. We can assume that we have the following pieces of information from the previous estimate: $\vec{x}^{apr}$ and $\boldsymbol{P}^{apr}$. Here $\vec{x}^{apr}$ is the update to the original state estimate $\vec{x}_0^{ref}$, similar to $\vec{x}$ above. associated covariance matrix.

Consider the following cost function

$$J = (\vec{x} - \vec{x}^{apr})^T \mathbf{\Lambda} (\vec{x} - \vec{x}^{apr}) + \vec{\rho}^T \vec{\rho}$$

Here $\mathbf{\Lambda}$ is the information matrix where $\mathbf{\Lambda} = (\boldsymbol{P}^{apr})^{-1}$. This new cost function penalizes deviations from the previous batch's estimate, weighed by $\mathbf{\Lambda}$, in addition to the original cost function as before ($\vec{\rho}^T \vec{\rho}$).

Due to the nature of an inverse covariance matrix, $\mathbf{\Lambda}$ is positive semi-definite and can be written as $\mathbf{\Lambda} = \boldsymbol{S}^T \boldsymbol{S}$. Rewriting the cost function, we can see

$$\begin{aligned} J &= (\vec{x} - \vec{x}^{apr})^T \mathbf{\Lambda} (\vec{x} - \vec{x}^{apr}) + (\vec{\xi} - \boldsymbol{A}\vec{x})^T (\vec{\xi} - \boldsymbol{A}\vec{x}) \\ &= \left(\begin{bmatrix} \boldsymbol{S}\vec{x}^{apr} \\ \vec{\xi} \end{bmatrix} - \begin{bmatrix} \boldsymbol{S} \\ \boldsymbol{A} \end{bmatrix} \vec{x} \right)^T \left(\begin{bmatrix} \boldsymbol{S}\vec{x}^{apr} \\ \vec{\xi} \end{bmatrix} - \begin{bmatrix} \boldsymbol{S} \\ \boldsymbol{A} \end{bmatrix} \vec{x} \right) \end{aligned}$$

Following Montenbruck and Gill's formulation [7], the solution for the minimum of the cost function can be found as

$$\vec{x}^{lsq} = \left(\begin{bmatrix} \boldsymbol{S} \\ \boldsymbol{A} \end{bmatrix}^T \begin{bmatrix} \boldsymbol{S} \\ \boldsymbol{A} \end{bmatrix} \right)^{-1} \begin{bmatrix} \boldsymbol{S} \\ \boldsymbol{A} \end{bmatrix}^T \begin{bmatrix} \boldsymbol{S}\vec{x}^{apr} \\ \vec{\xi} \end{bmatrix}$$

This simplifies to the following equation.

$$\vec{x} = (\mathbf{\Lambda} + \boldsymbol{A}^T \boldsymbol{A})^{-1} (\mathbf{\Lambda}\vec{x}^{apr} + \boldsymbol{A}^T \vec{\xi}) \tag{3.7}$$

where

$$(\boldsymbol{P})^{-1} = \mathbf{\Lambda} + (\boldsymbol{A}^T \boldsymbol{A}) \tag{3.8}$$

3.1.3 Final Formulation

Combining the two sections above into one set of equations is rather simple. To accomplish this, we must use the following relations.

$$\boldsymbol{Z} = \boldsymbol{A}^T \boldsymbol{W} \boldsymbol{A} = \boldsymbol{A}^T \boldsymbol{W}^{1/2} \boldsymbol{W}^{1/2} \boldsymbol{A} = \boldsymbol{B}^T \boldsymbol{B}$$

$$\frac{\partial \boldsymbol{Z}}{\partial \vec{x}} = 2\boldsymbol{B}^T \frac{\partial \boldsymbol{B}}{\partial \vec{x}} = 2\boldsymbol{B}^T \boldsymbol{W}^{1/2} \frac{\partial \boldsymbol{A}}{\partial \vec{x}}$$

The first relation is used to transform the weighted variation to a form identical to the original system, (3.4). This allows us to directly include *a priori* considerations without change. The second relation is utilized when finding the minimum of the cost function. Combining (3.6) and (3.7), we get out final definition for the update to the original state vector guess $\vec{x}$

$$\vec{x} = (\boldsymbol{\Lambda} + \boldsymbol{A}^T \boldsymbol{W} \boldsymbol{A})^{-1} (\boldsymbol{\Lambda} \vec{x}^{apr} + \boldsymbol{A}^T \boldsymbol{W} \vec{\xi}) \tag{3.9}$$

and consequently

$$(\boldsymbol{P})^{-1} = (\boldsymbol{P}^{apr})^{-1} + (\boldsymbol{A}^T \boldsymbol{W} \boldsymbol{A}) \tag{3.10}$$

It is noteworthy that the sum of $\boldsymbol{\Lambda}$ and $\boldsymbol{A}^T \boldsymbol{W} \boldsymbol{A}$ is required to be non-singular, while individually either can be singular. In pseudocode, the least squares filter takes the following form

Algorithm 1: Generic Batch Least Squares Filter

Result: $\vec{x}, \vec{x}_0, \boldsymbol{P}_0$
$\boldsymbol{A}$ = zeros()
$\vec{\xi}$ = zeros()
Given $\vec{x}_0$, $\boldsymbol{W}$, $\boldsymbol{\Lambda}$, $\vec{x}^{apr}$ **while** *stopping criteria not met* **do**
  for l *in number of observations* **do**
    find $\vec{x}_l$
    find $\boldsymbol{A}_l(\boldsymbol{x}), \vec{\xi}(\vec{x}_l)$
  end
  $\vec{x} = (\boldsymbol{\Lambda} + \boldsymbol{A}^T \boldsymbol{W} \boldsymbol{A})^{-1} (\boldsymbol{\Lambda} \vec{x}^{apr} + \boldsymbol{A}^T \boldsymbol{W} \vec{\xi})$
  $\vec{x}_0^{k+1} = \vec{x}_0^k + \vec{x}$ where $k = 1, 2, \ldots n$ until stopping criteria is met
end
$\boldsymbol{P}_0 = (\boldsymbol{\Lambda} + (\boldsymbol{A}^T \boldsymbol{W} \boldsymbol{A}))^{-1}$

3.2 Implementation

As with all implementations of pure theory, design decisions must be made. The following items must be addressed, there are multiple valid approaches for each.

1. Determination of the state vector
2. Calculation of partial derivative matrix: numerical versus analytical
3. Propagation method

4. Matrix Storage
5. Matrix inversion
6. Stopping criteria

3.2.1 State Vector

The state vector is the language of the least squares filter. The filter constantly updates it and returns values that all are directly related to the state vector. How we chose to define that vector can have significant impacts on the execution and accuracy of the algorithm. When prototyping, two sets were considered and each implemented.

$$\vec{x} = \begin{bmatrix} r_x \\ r_y \\ r_z \\ v_x \\ v_y \\ v_z \end{bmatrix} \text{ and } \quad \vec{x} = \begin{bmatrix} a \\ e \\ i \\ \Omega \\ \omega \\ \theta \end{bmatrix}$$

Between these two sets, there is one that is more advantageous in a significant manner. By defining an orbit via classical orbital elements, we are prone to singularities. Consider the circumstance where the orbit is perfectly equatorial and $\vec{e}$ is $\vec{0}$. Changes in the right ascension of the ascending node (Ω) are indistinguishable from changes in the argument of periapsis (ω) or the true anomaly (θ). As a result, when implementing this project, the state vector was defined in terms of $\vec{r}$ and $\vec{v}$ so that a singularity would never occur. However, towards the end of the project, I found another set of orbital elements with only one singularity [14]. Since this information came so late in the project; it was not feasible to implemented them. However, they are conceptually interesting and I will describe them here.

Consider the following set of modified equinoctial elements,

$$\vec{x} = \begin{bmatrix} p \\ f \\ g \\ h \\ k \\ L \end{bmatrix} = \begin{bmatrix} a(1-e^2) \\ e\cos(\omega+\Omega) \\ e\sin(\omega+\Omega) \\ \tan(i/2)\cos(\Omega) \\ \tan(i/2)\sin(\Omega) \\ \Omega+\omega+\theta \end{bmatrix}$$

where

$$
\begin{aligned}
p &= \text{semi-latus rectum} \\
a &= \text{semi-major axis} \\
e &= \text{eccentricity} \\
i &= \text{inclination} \\
L &= \text{true longitude}
\end{aligned}
$$

In the ECI (Earth Centered Inertial) frame, the position vector is

$$
\vec{r} = \begin{bmatrix} \frac{r}{s^2}(\cos L + \alpha^2 \cos L + 2hk \sin L \\ \frac{r}{s^2}(\sin L - \alpha^2 \cos L + 2hk \cos L \\ \frac{2r}{s^2}(h \sin L - k \cos L \end{bmatrix}
$$

and velocity vector is

$$
\vec{v} = \begin{bmatrix} -\frac{1}{s^2}\sqrt{\frac{\mu}{p}}(\sin L + \alpha^2 \sin L - 2hk \cos L + g - 2fhk + \alpha^2 g) \\ -\frac{1}{s^2}\sqrt{\frac{\mu}{p}}(-\cos L - \alpha^2 \cos L + 2hk \sin L - f + 2ghk + \alpha^2 f) \\ \frac{2}{s^2}\sqrt{\frac{\mu}{p}}(h \cos L + k \sin L + fh + gk) \end{bmatrix}
$$

where

$$
\begin{aligned}
\alpha^2 &= h^2 - k^2 \\
s &= 1 + h^2 + k^2 \\
r &= \frac{p}{w} \\
w &= 1 + f + \cos L + gsinL
\end{aligned}
$$

These values vary less than cartesian $\vec{r}$ and $\vec{v}$ coordinates, and are considered to be more stable. I have not confirmed this yet and would be interested in seeing their impact. It is important to note that a singularity does occur when $i = \pm 180\,\text{deg}$ as

$$
\tan(\frac{i}{2}) = \tan \pm 90 = \pm\infty
$$

Notably, $\vec{r}$ and $\vec{v}$ possess no singularities.

3.2.2 State Transition Matrix

The state transition matrix ($\mathbf{\Phi}_{t_0}^{t_1}$) can be used to move a state from time t_0 to t_1 using the following relation.

$$\vec{x}_1 = \mathbf{\Phi}_{t_0}^{t_1}\vec{x}_0$$

While we will propagate states forward using integration, this matrix will prove useful later. It is helpful to consider how to build it. We will closely follow Montenbruck's and Gill's [7] explanation. Let's consider the state vector

$$\vec{x}_t = \begin{bmatrix}\vec{r}\\ \vec{v}\end{bmatrix}$$

taking a time derivative, we see that

$$\frac{d}{dt}\vec{x}_t = f(t,\vec{x}_0) = \begin{bmatrix}\vec{v}(t)\\ \vec{a}(t,\vec{r},\vec{v})\end{bmatrix} \tag{3.11}$$

Before moving on, it is relevant to point out that the state transition matrix $\mathbf{\Phi}$ is defined as follows.

$$\mathbf{\Phi}_{t_0}^{t} \equiv \frac{\partial \vec{x}_t}{\partial \vec{x}_0} \tag{3.12}$$

$\vec{x}_t$ corresponds to the state at epoch t_k and x_0 at epoch t_0.

Taking a partial derivative of (3.11), we see that

$$\frac{\partial}{\partial \vec{x}_0}\frac{d}{dt}\vec{x}_t = \frac{\partial f(t,\vec{x}_0)}{\partial \vec{x}_0} = \frac{\partial f(t,\vec{x}_0)}{\partial \vec{x}_t}\frac{\partial \vec{x}_t}{\partial \vec{x}_0} = \frac{\partial f(t,\vec{x}_0)}{\partial \vec{x}_t}\mathbf{\Phi}_{t_0}^{t}$$

Swapping the order of derivatives, we see that

$$\frac{d}{dt}\mathbf{\Phi}_{t_0}^{t} = \frac{\partial f(t,\vec{x}_0)}{\partial \vec{x}_t}\mathbf{\Phi}_{t_0}^{t}$$

more explicitly,

$$\frac{d}{dt}\mathbf{\Phi}_{t_0}^{t} = \begin{bmatrix}\mathbf{0}_{3\text{x}3} & \boldsymbol{I}_{3\text{x}3}\\ \dfrac{\partial \vec{a}(t,\vec{r},\vec{v})}{\partial \vec{r}(t)} & \dfrac{\partial \vec{a}(t,\vec{r},\vec{v})}{\partial \vec{v}(t)}\end{bmatrix}\mathbf{\Phi}_{t_0}^{t} \tag{3.13}$$

In (3.13), we see an explicit need to find partial derivatives of the acceleration due to all forces with respect to position and velocity. While perturbing forces will be discussed more directly in the next chapter, it is worth noting that some perturbing forces are incredibly non-linear,

to evaluate $\mathbf{\Phi}$ analytically, they must be linearized. This in turn can significantly limit the accuracy of the model. To accurately find $\mathbf{\Phi}$, we will instead use numerical methods.

The most effective numerical derivative scheme was found to be the second-order centered difference equation.

$$\frac{\partial f(t, \vec{x}_0)}{\partial \vec{x}_i} = \frac{f(t, \vec{x}_0 + \Delta\vec{x}_i) - f(t, \vec{x}_0 - \Delta\vec{x}_i)}{2\Delta\vec{x}_i} + O(\Delta\vec{x}_i^2)$$

Using (3.12), the function f is the propagated state at t_k. $\Delta\vec{x}_i$ corresponds to one dimension of the state vector. Since positional coordinates and velocities exist on such separate scales, it was helpful to pick a constant $\Delta\vec{x}_i$ and $\Delta\vec{v}_i$. These values were the same in all three directions, to not prioritize accuracy between polar/equatorial orbits. While multiple values were tested, the following were found to be in the middle of a large acceptable range. $\Delta\vec{x}_i = 1$ km and $\Delta\vec{v}_i = .05$ km/s.

3.2.3 Partial Derivative Matrix

As shown above, the partial derivative matrix ($\boldsymbol{A}$) is fundamental to any least squares filter. Repeating it's definition,

$$\boldsymbol{A} \equiv \frac{\vec{\xi}}{\vec{x}}$$

Much like $\mathbf{\Phi}$, theoretically $\boldsymbol{A}$ can be calculated analytically or numerically. The analytical approach would require linearizing the observation function which would be possible if the observer was geocentric. Given the distinct lack of telescopes at the Earth's core, we have opted to calculate $\boldsymbol{A}$ numerically. Much like $\mathbf{\Phi}$, we utilized the second order centered difference equation. In this case, f is the observation function and $\Delta\vec{x}_i$ is an element of the current estimate of the state vector.

Calculating $\boldsymbol{A}$ numerically does contribute to the computational cost of the algorithm. The centered difference equation requires 2 function calls for each column in the m x n matrix. For our implementation, where $m = 6$ for each element of position and velocity, this totals to 12 function calls for each observation for every iteration of the outer loop of the algorithm. As a side note, truncating the Taylor expansion of $\vec{\rho}$ after the second order term in (3.2) would require an additional 24 function calls per observation per iteration. This triples our computation time for minimal improvement. In the case where the observation function mimics a radar system (range, azimuth, elevation), first order expansion requires 18 function calls, with the second order requiring 54. This is 4 times increase in computational cost. It was found that propagation function calls were the primary source of computing time, therefore reducing these as much as possible would be valuable to the user, so long as the implementation is sufficiently stable.

3.2.4 Covariance Propagation

As we identified above, we have elected to propagate state numerically. Unlike state propagation, covariance propagation explicitly relies upon the state transition matrix. This subsection explores $\boldsymbol{\Phi}$ and covariance propagation. Lastly, this exercise is independent of the LSQ filter logic, but is relevant to the implementation as missions are typically interested in uncertainty in the future.

The covariance matrix at time t_0 is defined by the following relation

$$\boldsymbol{P}_0 = \left[\frac{\partial \vec{x}_0}{\partial \vec{\xi}}\right] \left[\frac{\partial \vec{x}_0}{\partial \vec{\xi}}\right]^T$$

At epoch t_k, the covariance matrix would be defined similarly

$$\boldsymbol{P}_t = \left[\frac{\partial \vec{x}_t}{\partial \vec{\xi}}\right] \left[\frac{\partial \vec{x}_t}{\partial \vec{\xi}}\right]^T$$

Using the chain rule, we can see that

$$\boldsymbol{P}_t = \left[\frac{\partial \vec{x}_t}{\partial \vec{x}_0}\right] \left[\frac{\partial \vec{x}_0}{\partial \vec{\xi}}\right] \left[\frac{\partial \vec{x}_0}{\partial \vec{\xi}}\right]^T \left[\frac{\partial \vec{x}_t}{\partial \vec{x}_0}\right]^T$$

Substituting in our definitions of $\boldsymbol{P}_0$ and $\boldsymbol{\Phi}_{t_0}^t$

$$\boldsymbol{P}_t = \boldsymbol{\Phi} \boldsymbol{P}_0 \boldsymbol{\Phi}^T \tag{3.14}$$

The same centered difference equation in the previous subsection can be used to find a numerical state transition matrix, which has the same side effect of being more accurate than an analytical one. However, the effects of perturbing forces often have a minimal impact on propagating the covariance matrix. Depending upon the mission, it may be acceptable to save on execution time by evaluating $\boldsymbol{\Phi}$ as described above where no perturbing forces are included. This required implementing significant support for a non-issue for our product. The process of evaluating the state transition matrix numerically was not painful and all perturbations included in the state propagation are extended to the covariance propagation.

3.2.5 Matrix Storage

Across the multiple references used in this project, only one included this clever spin on implementation. On purpose, little has been said about the size of the matrices in equations (3.9) and (3.10). Remember, the residual observation vector $\vec{\rho}$ is of length n by 1, where

n is dependent upon the the number of observable values, p, and the l, the number of observations. Explicitly $n = p * l$. Additionally, in the pseudocode description of the generic algorithm states "find $\boldsymbol{A}_l(\vec{x}), \vec{\xi}(\vec{x}_l)$" and does not relate these submatrices to their larger relatives. Let's do that now.

$$\boldsymbol{A}_{nxm} = \begin{bmatrix} \boldsymbol{A}_1 \\ \boldsymbol{A}_2 \\ \vdots \\ \boldsymbol{A}_l \end{bmatrix} \quad \boldsymbol{W}_{nxn} = diag\left(\begin{bmatrix} \boldsymbol{W}_1 & \boldsymbol{W}_2 & \dots & \boldsymbol{W}_l \end{bmatrix}\right) \quad \vec{\xi}_{nx1} = \begin{bmatrix} \vec{\xi}_1 \\ \vec{\xi}_2 \\ \vdots \\ \vec{\xi}_l \end{bmatrix}$$

Together,

$$\boldsymbol{A}^T_{mxn}\ \boldsymbol{W}_{nxn}\ \boldsymbol{A}_{nxm} = \mathbb{M}_{mxm}$$

Similarly,

$$\boldsymbol{A}^T_{mxn}\ \boldsymbol{W}_{nxn}\ \vec{\xi}_{nx1} = \mathbb{M}_{mx1}$$

Where $\mathbb{M}$ is a generic matrix. Consider the circumstance where there are two observable values, right ascension (α) and declination (δ), and 1000 observations. This gives $n = 2000$. Additionally, lets assume our model parameter vector is exclusively the state vector and we are not considering any other parameters, m=6. The above system becomes

$$\boldsymbol{A}^T_{6x2000}\ \boldsymbol{W}_{2000x2000}\ \boldsymbol{A}_{2000x6} = \mathbb{M}_{6x6}$$

$$\boldsymbol{A}^T_{6x2000}\ \boldsymbol{W}_{2000x2000}\ \vec{\xi}_{2000x1} = \mathbb{M}_{6x1}$$

As this clearly demonstrates, $\boldsymbol{A}$, $\boldsymbol{W}$, and $\vec{\xi}$ are all dependent upon l and can get quite large and grow with the number of observations. If we consider their structure and how they are multiplied together, we can parallelize some of the algebra to store matrices with a constant size. Let's consider the circumstance where $p = 2$, $l = 2$, and $m = 6$. For each observation, we have α_l and δ_l.

$$\boldsymbol{A}_{4x6} = \left[\begin{array}{cccccc} \frac{\partial \alpha_1}{\partial r_x} & \frac{\partial \alpha_1}{\partial r_y} & \frac{\partial \alpha_1}{\partial r_z} & \frac{\partial \alpha_1}{\partial v_x} & \frac{\partial \alpha_1}{\partial v_y} & \frac{\partial \alpha_1}{\partial v_z} \\ \frac{\partial \delta_1}{\partial r_x} & \frac{\partial \delta_1}{\partial r_y} & \frac{\partial \delta_1}{\partial r_z} & \frac{\partial \delta_1}{\partial v_x} & \frac{\partial \delta_1}{\partial v_y} & \frac{\partial \delta_1}{\partial v_z} \\ \hline \frac{\partial \alpha_2}{\partial r_x} & \frac{\partial \alpha_2}{\partial r_y} & \frac{\partial \alpha_2}{\partial r_z} & \frac{\partial \alpha_2}{\partial v_x} & \frac{\partial \alpha_2}{\partial v_y} & \frac{\partial \alpha_2}{\partial v_z} \\ \frac{\partial \delta_2}{\partial r_x} & \frac{\partial \delta_2}{\partial r_y} & \frac{\partial \delta_2}{\partial r_z} & \frac{\partial \delta_2}{\partial v_x} & \frac{\partial \delta_2}{\partial v_y} & \frac{\partial \delta_2}{\partial v_z} \end{array}\right]$$

$$\boldsymbol{W}_{4\text{x}4} = \left[\begin{array}{cc|cc} \frac{1}{\sigma_{\alpha_1}^2} & & & \\ & \frac{1}{\sigma_{\delta_1}^2} & & \\ \hline & & \frac{1}{\sigma_{\alpha_2}^2} & \\ & & & \frac{1}{\sigma_{\delta_2}^2} \end{array}\right] \quad \vec{\xi}_{4\text{x}1} = \left[\begin{array}{c} \Delta\alpha_1 \\ \Delta\delta_1 \\ \hline \Delta\alpha_2 \\ \Delta\delta_2 \end{array}\right]$$

The lines above demonstrate where values impacted by a single observation are partitioned. When multiplying $\boldsymbol{A}^T\boldsymbol{W}\boldsymbol{A}$ we see

$$\left[\begin{array}{cc|cc} \frac{\partial\alpha_1}{\partial r_x} & \frac{\partial\delta_1}{\partial r_x} & \frac{\partial\alpha_2}{\partial r_x} & \frac{\partial\delta_2}{\partial r_x} \\ \frac{\partial\alpha_1}{\partial r_y} & \frac{\partial\delta_1}{\partial r_y} & \frac{\partial\alpha_2}{\partial r_y} & \frac{\partial\delta_2}{\partial r_y} \\ \frac{\partial\alpha_1}{\partial r_z} & \frac{\partial\delta_1}{\partial r_z} & \frac{\partial\alpha_2}{\partial r_z} & \frac{\partial\delta_2}{\partial r_z} \\ \frac{\partial\alpha_1}{\partial v_x} & \frac{\partial\delta_1}{\partial v_x} & \frac{\partial\alpha_2}{\partial v_x} & \frac{\partial\delta_2}{\partial v_x} \\ \frac{\partial\alpha_1}{\partial v_y} & \frac{\partial\delta_1}{\partial v_y} & \frac{\partial\alpha_2}{\partial v_y} & \frac{\partial\delta_2}{\partial v_y} \\ \frac{\partial\alpha_1}{\partial v_z} & \frac{\partial\delta_1}{\partial v_z} & \frac{\partial\alpha_2}{\partial v_z} & \frac{\partial\delta_2}{\partial v_z} \end{array}\right] \left[\begin{array}{cc|cc} \frac{1}{\sigma_{\alpha_1}^2} & & & \\ & \frac{1}{\sigma_{\delta_1}^2} & & \\ \hline & & \frac{1}{\sigma_{\alpha_2}^2} & \\ & & & \frac{1}{\sigma_{\delta_2}^2} \end{array}\right] \left[\begin{array}{cccccc} \frac{\partial\alpha_1}{\partial r_x} & \frac{\partial\alpha_1}{\partial r_y} & \frac{\partial\alpha_1}{\partial r_z} & \frac{\partial\alpha_1}{\partial v_x} & \frac{\partial\alpha_1}{\partial v_y} & \frac{\partial\alpha_1}{\partial v_z} \\ \frac{\partial\delta_1}{\partial r_x} & \frac{\partial\delta_1}{\partial r_y} & \frac{\partial\delta_1}{\partial r_z} & \frac{\partial\delta_1}{\partial v_x} & \frac{\partial\delta_1}{\partial v_y} & \frac{\partial\delta_1}{\partial v_z} \\ \hline \frac{\partial\alpha_2}{\partial r_x} & \frac{\partial\alpha_2}{\partial r_y} & \frac{\partial\alpha_2}{\partial r_z} & \frac{\partial\alpha_2}{\partial v_x} & \frac{\partial\alpha_2}{\partial v_y} & \frac{\partial\alpha_2}{\partial v_z} \\ \frac{\partial\delta_2}{\partial r_x} & \frac{\partial\delta_2}{\partial r_y} & \frac{\partial\delta_2}{\partial r_z} & \frac{\partial\delta_2}{\partial v_x} & \frac{\partial\delta_2}{\partial v_y} & \frac{\partial\delta_2}{\partial v_z} \end{array}\right]$$

Elementwise, the result of this product can be written as

$$\left[\boldsymbol{A}_{m\text{x}n}^T\ \boldsymbol{W}_{n\text{x}n}\ \boldsymbol{A}_{n\text{x}m}\right]_{ij} = \sum_{k=1}^{n} \boldsymbol{A}_{ik}^T \boldsymbol{W}_j \boldsymbol{A}_{kj}$$

where $n = p * l$ and $i, j = 1, 2, ..., m$. Refering back to the definitions of $\boldsymbol{A}_l$ and $\boldsymbol{W}_l$, we can see this is exactly equal to

$$\left[\boldsymbol{A}_{m\text{x}n}^T\ \boldsymbol{W}_{n\text{x}n}\ \boldsymbol{A}_{n\text{x}m}\right]_{ij} = \sum_{l} \sum_{k=1}^{p} \boldsymbol{A}_{lik}^T \boldsymbol{W}_k \boldsymbol{A}_{lkj}$$

Generalizing to matrix form, this is

$$\boldsymbol{A}_{m\text{x}n}^T\ \boldsymbol{W}_{n\text{x}n}\ \boldsymbol{A}_{n\text{x}m} = \sum_{l} \boldsymbol{A}_{m\text{x}p}^T\ \boldsymbol{W}_{p\text{x}p}\ \boldsymbol{A}_{p\text{x}m} \tag{3.15}$$

While not explicitly shown, this also holds true for the right-hand side vector, where

$$\boldsymbol{A}^T_{mxn}\ \boldsymbol{W}_{nxn}\ \vec{\xi}_{nx1} = \sum_l \boldsymbol{A}^T_{mxp}\ \boldsymbol{W}_{pxp}\ \vec{\xi}_{pxm} \tag{3.16}$$

It should be noted that while this does save on memory usage, this does not cut down on the execution time associated with matrix multiplication as $\boldsymbol{W}$ is a diagonal matrix. In summary, we have targeted the largest variable of memory storage and reduced it to a constant size defined by the state vector and observation vector lengths. This remains fundamentally different from processing each data point individually. This change will be represented in a future written algorithm at the end of the chapter.

3.2.6 Matrix Inversion

An important assumption made in an earlier section of this chapter was that $\boldsymbol{A}^T\boldsymbol{W}\boldsymbol{A}$ was invertible. While $\boldsymbol{\Lambda} + \boldsymbol{A}^T\boldsymbol{W}\boldsymbol{A}$ was discussed explicitly, we must consider the case where the user does not have *a priori* information ($\boldsymbol{\Lambda} = \boldsymbol{0}_{mxm}$). Unfortunately our product, $\boldsymbol{\Pi} = \boldsymbol{A}^T\boldsymbol{W}\boldsymbol{A}$ is always near singular, if not singular. The built-in inverse function native to SciPy and NumPy are unhelpful. In the rare case when they do not throw an error, the answer is far from satisfying the relation $\boldsymbol{\Pi}^{-1}\boldsymbol{\Pi} = \boldsymbol{I}$. In one test, the residual of $\boldsymbol{\Pi}^{-1}\boldsymbol{\Pi} - \boldsymbol{I}$ had a norm on the order of $1e17$.

Similarly, when prototyping in MATLAB with a simplified two-body gravitational model, I encountered a near-singular warning. However, unlike SciPy, MATLAB was able to invert the matrix appropriately. Before we discuss the working solution, I would like to explore inversion techniques suggested in my supporting documents [15] [13] [7] . The most often recommended solutions include taking advantage of factoring our target matrix product into a set of orthogonal and non-orthogonal matrices, and allowing the orthogonal matrices to be eliminated via the product $\boldsymbol{B}^T\boldsymbol{B}$. For example, QR factorization, Householder Transformation, Givens Rotations, Cholesky decomposition, or singular value decomposition are all commonly referenced. To apply these algorithms, we must use the previously mentioned transformation

$$\boldsymbol{A}^T\boldsymbol{W}\boldsymbol{A} = \boldsymbol{B}^T\boldsymbol{B}$$

where

$$\boldsymbol{B} = \boldsymbol{W}^{1/2}\boldsymbol{A}$$

Finding the square-root of a matrix can be quite costly because it must be conducted for every iteration the main loop requires. In "Statistical Orbit Determination" [13] Tapley, Schutz, and Born provide square-root free versions of the cholesky, QR, Givens Transformation algorithms. In order to implement these in the project as this point, the algorithms would have to be written in python, a high-level programming language. As the previous paragraph

indicates, I found an alternate I believe to be a better replacement due to it's high accuracy and quick execution time.

When prototyping in MATLAB, I noticed that the native "\" operator was able to solve the system described in (3.4), repeated below for clarity.

$$(\boldsymbol{A}^T\boldsymbol{A})\boldsymbol{x} = \boldsymbol{A}^T\vec{\xi}$$

As we are now discussing solving the normal equation, I will switch notation for the remainder of this section. Notably

$$\boldsymbol{A}^T\boldsymbol{W}\boldsymbol{A} \Rightarrow \boldsymbol{A} \quad \text{and} \quad \boldsymbol{A}^T\boldsymbol{W}\vec{\xi} \Rightarrow \vec{b}$$

such that (3.4) becomes

$$\boldsymbol{A}x = b$$

Additionally. While vector were previously indicated with an arrow, here they are in normal italics, without a special indicator. Matrices will continue to be bold.

Upon further investigation into MATLAB's inner working, I was able to learn that it was using the banded_solver function, a wrapper for the LAPACK (Linear Algebra Pack) function dgbsv. Notably, this function requires converting a "banded" matrix into the diagonalized normal form, very different from the Jordan normal form. To accomplish this, we must first know the upper and lower bandwidths of our target "banded" matrix $\boldsymbol{A}$. The bandwidth of a banded matrix is used as a measure of how far away the zeros are from the main diagonal. What makes this more confusing is that our matrix $\boldsymbol{A}$ does not resemble a banded matrix. It is merely symmetrical. Using MATLAB's built-in bandwidth function, I was able to determine that our 6 x 6 example had 5 for both bandwidths. This held true across every iteration.

Converting $\boldsymbol{A}$ into the diagonalized normal form $\boldsymbol{ab}$ was done using a sample function found on stack exchange. It was verified against the few examples I could find online. See the following example, from the SciPy banded_solver support page. The upper bandwidth is 2 and the lower bandwidth is 1.

$$\boldsymbol{A} = \begin{bmatrix} 5 & 2 & -1 & 0 & 0 \\ 1 & 4 & 2 & -1 & 0 \\ 0 & 1 & 3 & 2 & -1 \\ 0 & 0 & 1 & 2 & 2 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix} \Rightarrow \boldsymbol{ab} = \begin{bmatrix} 0 & 0 & -1 & -1 & -1 \\ 0 & 2 & 2 & 2 & 2 \\ 5 & 4 & 3 & 2 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$

This sort of transformation maintains the vital information while condensing the matrix. Unfortunately, I could find very little material validating the name of this new matrix structure, let alone how to generate it.

The inputs for the banded_solver function include $\boldsymbol{ab}$, b, u and l, where u and l represent the upper and lower bandwidths. With all of the inputs ready, we solved the system and measured the residual. For this sub section alone, ρ represents the residual of inversions or solved systems.

$$\boldsymbol{A}x - b = \rho$$

If the system was solved perfectly, ρ would be exactly equal to 0. Due to normal computational errors, it will not be, but can be very close. While the residual changes with each iteration for any batch, its norm was found to be on the order of $1e-17$, sufficiently close to zero.

It is now time to draw an important distinction. This subsection is titled "Matrix Inversion" not solving a system of equations. As the name and inputs may imply, banded_solver was built to solve a system of equation. Let us consider the following system.

$$\boldsymbol{A}x = \boldsymbol{I}$$

then

$$x = \boldsymbol{A}^{-1}$$

Additionally, as previously established

$$\boldsymbol{P} = (\boldsymbol{A}^T \boldsymbol{W} \boldsymbol{A})^{-1}$$

and we must properly invert $\boldsymbol{P}$. This process is independent of the right-hand side vector b. Fortunately, as the above system implies, if we send an identity matrix instead of the right-hand side vector, the result from the banded_solver function is the inverse. To test the accuracy of this process we define a new residual, this time a matrix.

$$\boldsymbol{P}\boldsymbol{P}^{-1} - \boldsymbol{I} = \boldsymbol{\rho}$$

The norm of $\boldsymbol{\rho}$ was on the order of $1e-17$ as well. It is clear this is a viable approach for inversion. Additionally, by using the SciPy wrapper for the LAPACK dgbsv, the bulk of the numerical stress is being run in FORTRAN. Since Python is a much higher-level language, no algorithm written in Python can execute as quickly. Secondly, LAPACK is a well-known and high-quality product.

3.2.7 Stopping Criteria

There is no definitively correct way to handle stopping criteria. In "The Asteroid Identification Problem" [6], Andrea Milani describes least squares filters as a "pseudo-newton"

root solving method. Similarly, in "Statistical Orbit Determination" [13], Tapley, Schutz, and Born show that it is closely related to the Newton-Raphson method. As such, Tapley, Schutz, and Born suggest using the simplest possible criteria, $\vec{x} < \sigma$ where σ is some arbitrary value.

In his paper, Milani suggest the following criteria,

$$\vec{x}_k \dot{(} \boldsymbol{A}^T \boldsymbol{W} \boldsymbol{A})_k \vec{x}_k < \sigma$$

where sigma is a constant. He suggests finding an arbitrary value of the scalar σ that works. This expands upon the stopping criteria mentioned above by weighing it with the $(\boldsymbol{A}^T \boldsymbol{W} \boldsymbol{A})$ matrix.

Both of these are arbitrary as they just require the iteration's state update to be under a preset threshold. It doesn't relate to a natural convergence point. In Vallado's "Fundamentals of Astrodynamics and Applications" [15], he suggests defining a RMS value.

$$RMS = \sqrt{\vec{\xi}^T \boldsymbol{W} \vec{\xi}}$$

Comparing the current iteration's RMS value to the previous can determine how close the update is to convergence. For example, the program will continue while the following relation is true.

$$\left| \frac{RMS_{old} - RMS_{new}}{RMS_{old}} \right| \leq \epsilon$$

Here we can set ϵ to any value between 0 and 1. For the purposes of this project, I have chosen .9. The higher the value, the longer the algorithm will take to converge. Since we are looking for maximum accuracy, this allows us to approach the natural convergence point and not pre-maturely end the algorithm once an arbitrary norm has been met.

3.2.8 Final Implemented Algorithm

The previous algorithm was a generalized batch least squares filter. The following algorithm considers some of the decisions mentioned in the previous subsections and serves as a

summary of this chapter.

Algorithm 2: Implemented Batch Least Squares Filter

Result: $\vec{x}_0, \vec{x}, \boldsymbol{P}_0$
$\boldsymbol{C}$ = zeros(6, 6)
$\vec{D}$ = zeros(6, 1)
Given $\vec{x}_0$, $\boldsymbol{W}$, $\boldsymbol{\Lambda}$, $\vec{x}^{apr}$ **while** *stopping criteria (RMS_{old}, RMS_{new}) not met* **do**
  for *l in number of observations* **do**
    find $\vec{x}_l$
    find $\boldsymbol{A}_l(\vec{x}_l), \vec{\xi}(\vec{x}_l)$
    $\boldsymbol{C} += \boldsymbol{A}_l^T \boldsymbol{W}_l \boldsymbol{A}_l$
    $\vec{D} += \boldsymbol{A}_l^T \boldsymbol{W}_l \vec{\xi}_l$
  end
  $\vec{x} = (\boldsymbol{\Lambda} + C)^{-1}(\boldsymbol{\Lambda}\vec{x}^{apr} + \vec{D})$
  $\vec{x}_0^{k+1} = \vec{x}_0^k + \vec{x}$ where $k = 1, 2, \ldots n$ until stopping criteria is met
  update RMS values
end
$\boldsymbol{P}_0 = (\boldsymbol{\Lambda} + \boldsymbol{C})^{-1}$

The following list details the remaining design decisions not included in the above algorithm:

1. $\vec{x}_t$ is found using Cowell's method
2. $\boldsymbol{A}$ is found numerically differentiating using centered difference equation
3. Invert $(\boldsymbol{\Lambda} + \boldsymbol{C})$ using banded_solver
4. The observation function inherent in $\vec{\xi}$ is in terms of right ascension (α) and declination (δ)
5. The value of ϵ in stopping criteria function is .9

3.3 Covariance Analysis

While a converged state vector is helpful, the covariance matrix is required to provide the full context. Let us consider one application for a least squares filter: the orbital debris problem. For a spacecraft to effectively apply course corrections to avoid incoming debris, it must not only know the most likely trajectory the debris is on, but also the entire "cloud" it could occupy. For three-dimensional space, the rule of thumb is to consider the volume defined by three sigmas. The following table by Vallado [15] makes a compelling argument for why.

Table 3.1: Probability an object is found within number of standard deviations by spatial dimensions. This assumes the errors in each dimension are uncorrelated.

Dimension(N)	1σ	2σ	3σ	4σ
1	.6827	.9545	.9973	.9999
2	.3935	.8647	.9889	.9996
3	.1987	.7385	.9707	.9989

Repeating (3.10),

$$(\boldsymbol{P})^{-1} = (\boldsymbol{P}^{apr})^{-1} + (\boldsymbol{A}^T \boldsymbol{W} \boldsymbol{A})$$

we can see that the covariance matrix of any batch is dependent upon the *a priori* covariance matrix and the product $\boldsymbol{A}^T \boldsymbol{W} \boldsymbol{A}$. This product is a symmetric, positive-definite matrix. At least under the circumstance where the state vector is in terms of $\boldsymbol{r}$ and $\boldsymbol{v}$, we can more easily understand the meaning behind its values. Let us consider it structure. We will start by defining $\boldsymbol{A}$ in block matrix form.

$$\boldsymbol{A} = \begin{bmatrix} \frac{\partial \vec{\xi}}{\partial \vec{r}} & \frac{\partial \vec{\xi}}{\partial \vec{v}} \end{bmatrix}$$

Next, we consider the product $\boldsymbol{A}^T \boldsymbol{A}$,

$$\boldsymbol{A}^T \boldsymbol{A} = \begin{bmatrix} \frac{\partial \vec{\xi}}{\partial \vec{r}} \\ \frac{\partial \vec{\xi}}{\partial \vec{v}} \end{bmatrix} \begin{bmatrix} \frac{\partial \vec{\xi}}{\partial \vec{r}} & \frac{\partial \vec{\xi}}{\partial \vec{v}} \end{bmatrix} = \begin{bmatrix} \left(\frac{\partial \vec{\xi}}{\partial \vec{r}}\right)^2 & \frac{\partial \vec{\xi}}{\partial \vec{r}} \frac{\partial \vec{\xi}}{\partial \vec{v}} \\ \frac{\partial \vec{\xi}}{\partial \vec{v}} \frac{\partial \vec{\xi}}{\partial \vec{r}} & \left(\frac{\partial \vec{\xi}}{\partial \vec{v}}\right)^2 \end{bmatrix} = \begin{bmatrix} \boldsymbol{\Pi}_{11} & \boldsymbol{\Pi}_{12} \\ \boldsymbol{\Pi}_{12}^T & \boldsymbol{\Pi}_{22} \end{bmatrix}$$

Here we can see that off-diagonal blocks are transposes of one another while the first and last blocks are independent of one another and depend entirely upon their respective magnitudes of the partials of $\boldsymbol{\xi}$. In the scenarios I ran, I found that the first quadrant was orders of magnitude smaller than its neighbors roughly double the gap with the last quadrant. This is relevant because the covariance matrix follows a similar but inverted pattern.

$$\boldsymbol{P} = \boldsymbol{\Pi}^{-1} = \begin{bmatrix} \boldsymbol{P}_{11} & \boldsymbol{P}_{12} \\ \boldsymbol{P}_{12}^T & \boldsymbol{P}22 \end{bmatrix}$$

Here, the first quadrant was much larger than the others, particularly the last quadrant. This gives a larger uncertainty in position, rather than velocity as expected. Additionally, the larger the values in $\boldsymbol{\Pi}$, the smaller the values in $\boldsymbol{P}$.

The least squares filter does not minimize variances. Through the definition of the cost function, the residual observational values are minimized. The covariance matrix is merely

an output. However, there are ways to influence the covariance matrix of an established system: quantity of measurements, quality of measurements, incorporating a previous batch (effectively the other two combined).

Quantity of measurements is the simplest approach to reduce uncertainty. Unfortunately, this has a relatively weak affect. If we consider the $\mathbf{\Pi}_l$ matrix, it is safe to assume it is close to the same magnitude for two separate observations with the same uncertainties near the same point in the orbit. By doubling the number of observations, we have doubled the entries of $\mathbf{\Pi}$ and halved the entries of $\boldsymbol{P}$. This has the impact of reducing the variances by $\sqrt{2}$. For double the work and computational effort, we have less than halved the variances. Often, in large pools of observations with varying uncertainties, there are a select number of highly accurate measurements that pull the error down, while the rest have little to no weight.

In the block definition of $\mathbf{\Pi}$ above, the weighted matrix was purposefully left out for the sake of simplicity. To expand further, we consider the definition repeated below.

$$\boldsymbol{W} = \begin{bmatrix} \frac{1}{\sigma_1^2} & 0 & 0 & 0 \\ 0 & \frac{1}{\sigma_2^2} & 0 & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & \frac{1}{\sigma_n^2} \end{bmatrix}$$

Let's consider 2 observations, where the uncertainties in measured values are $[1, 1]$ and $\left[\frac{1}{2}, \frac{1}{2}\right]$. Next, we build our two weighted matrices.

$$\boldsymbol{W}_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad \boldsymbol{W}_2 = \begin{bmatrix} 4 & 0 \\ 0 & 4 \end{bmatrix}$$

If we assume the observations are near one another, we can assume that $\boldsymbol{A}_1 = \boldsymbol{A}_2$. This gives $\boldsymbol{P}_2 = 4\boldsymbol{P}_1$, meaning the impact on uncertainty by the second observation is twice as strong as the first. Generalizing this, if observation A is n times more accurate than observation b, it has the same impact on the covariance matrix as n observations of comparable accuracy to observation b. This verifies the premise that a small number of highly accurate measurements can dominate many poor measurements.

3.3.1 Designing a System

One of the most significant lessons I learned through this project was how system design can affect the covariance output. While attempting to verify my project was working correctly,

I mimicked Vallado's example on page 770 [15]. The observation function he used included range in addition to two angular measurement. Compared to my purely two angles observation function, his covariance matrix was many orders of magnitude smaller. Initially, I assumed this was due to the nature of adding a third measurement, however, upon further reflection, I realized that this was due to a high precision of range measurements compared to angles.

Let's return our attention to (3.15), modified slightly below.

$$\left[\boldsymbol{A}^T_{mxn}\ \boldsymbol{W}_{nxn}\ \boldsymbol{A}_{nxm}\right]_{ij} = \sum_{l}\sum_{k}^{p} \frac{\boldsymbol{A}^T_{lik}\boldsymbol{A}_{lkj}}{\sigma_k^2}$$

Here we see two summations. The first is over number of observations, validating our discussion on quantity of data. The second is over p, the number of measurements per observation. If all of the partial derivatives and uncertainties are of the same order, then having three observations with two values is just as effective as having two observations with three values. This has the effect of dividing one observation, say by a radar system, into three separate observations. The resulting lesson is that quality is king. When designing a situational awareness system, any of the following measurements could be made to help identify an object's trajectory:

Position - GPS
Range - RADAR
Angles - Optics/RADAR
Doppler shift - RADAR/Optical Communications
Brightness - Telescope
Magnetic fields - On-board Magnetometer

If we look at the modified (3.15) closer, we see two components at play. $\boldsymbol{A}_{lik}$ and $\boldsymbol{A}_{lkj}$, which are driven by the target and σ_k is user-defined. To achieve a desired accuracy, two things must be considered. The first being the orbit itself, which drives the $\boldsymbol{A}$ matrix. Secondly, the next parameter to consider would be the uncertainty of the measured values (σ_p). This would give a baseline of how many observations would be required to drive a sufficiently low covariance matrix. Evaluating which combination or individual system would be optimal would depend upon the their ratio in the modified (3.15) and frequency of observations.

Chapter 4

State Propagation

The least squares filter requires predicted observations at epochs corresponding to those of the actual observations. Before we can do this, we must first propagate the state from an initial estimate to the corresponding epochs. This chapter explores the orbital mechanics required to move an Earth orbiting satellite through time. Naturally, we start with Newton's second law.

$$\ddot{\vec{r}} = -\frac{\mu}{r^3}\vec{r} + \vec{a}_D \tag{4.1}$$

In the circumstance where $\vec{a}_D = 0$, this system is described by the trajectory equation or can be solved algebraically through the Lagrange-Gibbs solution. Both descriptions break down once perturbations are included. To maximize physical accuracy, all perturbations should be considered. Given the varying orders of magnitude among perturbations, we will eventually impose a limit.

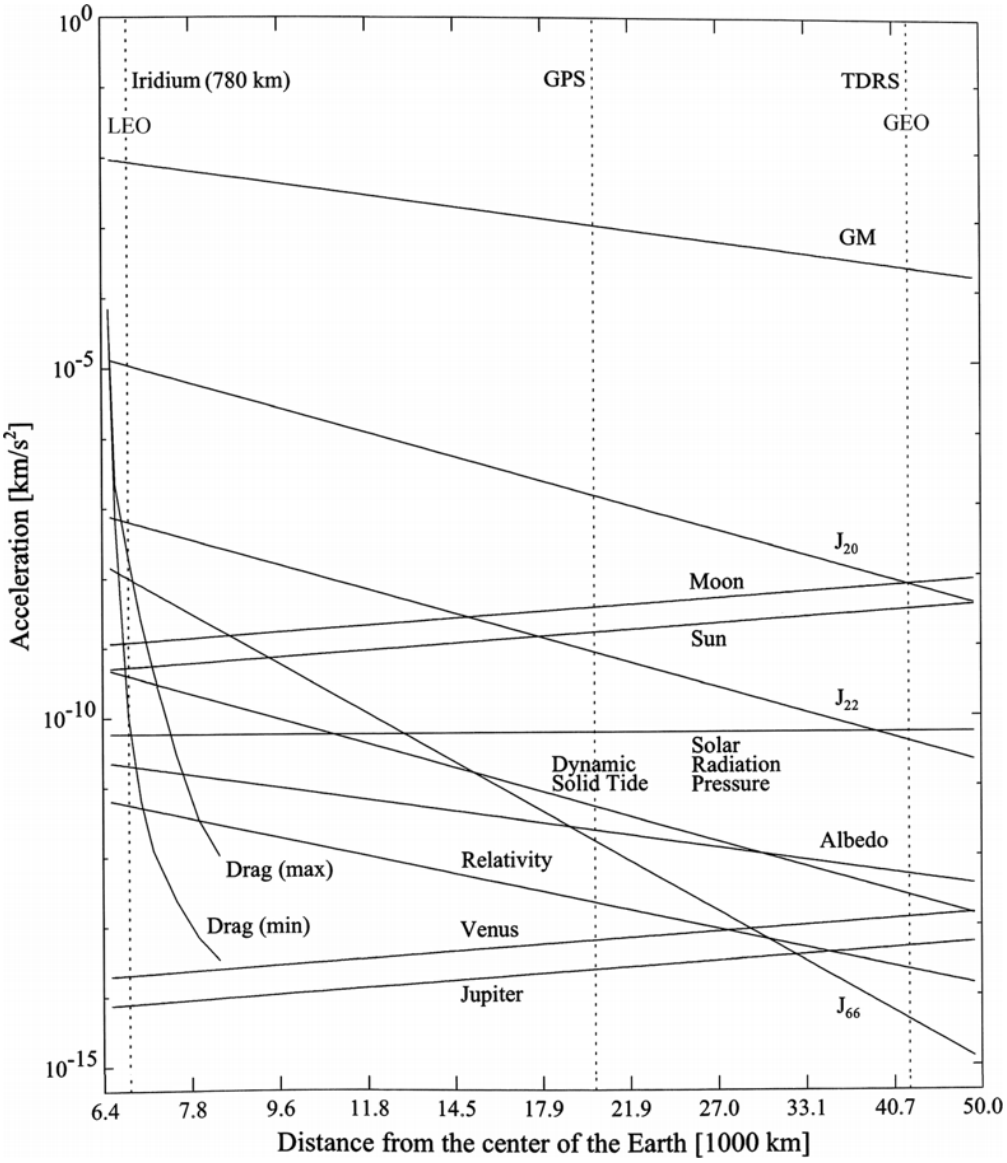

Figure 4.1: Order of magnitude plot for all perturbing accelerations as a function of altitude from Low Earth Orbit to Geostatonary orbits [7].

As we can see from 4.1, it is possible to describe perturbations 11 orders of magnitude less than the gravitational force of the Earth. The strongest perturbation, J_{20}, is $3-4$ order of magnitude weaker than Earth's gravity . Drag is particularly relevant at low altitudes but quickly dies off with altitude. Following J_{20}, we see J_{22}, lunar and solar gravities, solar radiation pressure, dynamic solid tide, albedo, and so forth. For the implementation of this project, we chose to cut-off our perturbing forces at $10^{-10}N$ and include only solar radiation pressure.

Traditionally, more precise models are not required. Only missions requiring particularly

high accuracy include perturbations beyond the ones included above. The most well-known exception would be the TOPEX/POSEIDON mission which required a radial position error less than 10 cm [8]. In addition to our perturbations, CNES/NASA considered Earth radiation pressure (Albedo), dynamic solid tide, and relativistic forces. Extending this project to include these perturbations is entirely possible and will be discussed in the Poliastro subsection.

4.1 Perturbation Methods

The two main ways of handling perturbing forces include Enke's method and Cowell's method. Enke's method involves solving the system initially as a two-body problem and then integrating the effects of perturbing forces on the motion separately. This method is less straight forward than its alternative and can be particularly complex as many perturbing forces are included. Cowell's method is incredibly simple, define $\bar{a}_D$ and integrate (4.1).

Integration can be a complex task between the many schemes and supporting techniques such as step-size control. There are a number of high-quality FORTRAN methods that can be used out of the box. One such optimal method is DOPRI8, developed by Prince and Dormand in 1981. This method is in the Runge-Kutta family of integrators and has solved the under-described set of equations inherent to the Runge-Kutta system to minimize the number of function calls with respect to accuracy in digits.

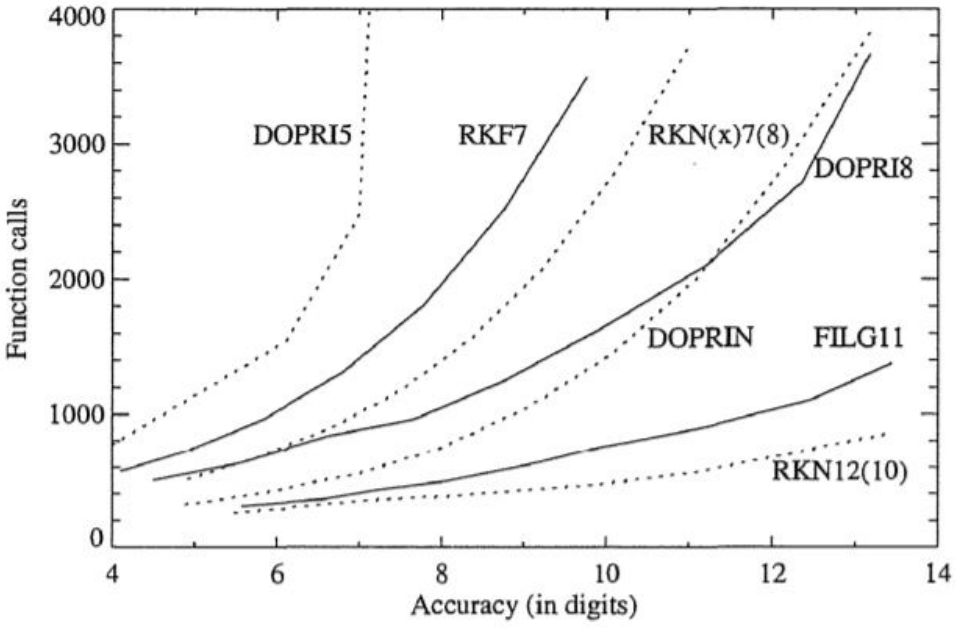

Figure 4.2: Comparison of integrators by Montenbruck and Gill [7]

While DOPRI8 is not the most accurate overall integrator in the plot above, it is a good tradeoff between accuracy and efficiency. FILG11 is a 11th-order RK integrator with 17 stages developed in 1987. DOPRIN is a RK 7th-order solver developed in 1987 that solves second order ODE's, much like (4.1). Lastly, RKN12(10) is a 12th-order solver, also developed by Dormand in 1987. RKN12(10) is available under the name D02LAF in the NAG library [7].

4.2 Force Models

4.2.1 Spherical Harmonics

Earth's gravitational field is much more complex than μ, the gravitational constant of the Earth, can capture. It is time-varying and not spherically symmetrical. While tides play a role in this, the most distinct feature is Earth's oblateness. While the radius of the Earth at the equator is $6378km$, the radius at the North Pole is $6356.75km$. The impact of Earth's oblateness is specifically referred to as J_2, which falls under the greater spherical harmonics' description.

Spherical Harmonics involves dividing the Earth into various zones with respect to latitude and longitude. These descriptions can become quite detailed. Some models included orders of 210. However, in 4.1, we see J_{20}, J_{22},and J_{66} explicitly. Poliastro handles only J_2 and J_3 with the following equations from [3]. Future expansion would be worthwhile. x, y, and z correspond to positions in the Earth-Centered Inertial Frame.

$$\vec{F}_{J_2} = \frac{3}{2}\frac{J_2 \mu R^2}{r^4}\left[\frac{x}{r}\left(5\frac{z^2}{r^2}-1\right)\vec{i} + \frac{y}{r}\left(5\frac{z^2}{r^2}-1\right)\vec{j} + \frac{z}{r}\left(5\frac{z^2}{r^2}-3\right)\vec{k}\right] \tag{4.2}$$

$$\vec{F}_{J_3} = \frac{1}{2}\frac{J_3 \mu R^3}{r^5}\left[5\frac{x}{r}\left(7\left(\frac{z}{r}\right)^3 - 3\frac{z}{r}\right)\vec{i} + 5\frac{y}{r}\left(7\left(\frac{z}{r}\right)^3 - 3\frac{z}{r}\right)\vec{j} + 3\left(\frac{35}{3}\left(\frac{z}{r}\right)^4 - 10\left(\frac{z}{r}\right)^2 + 1\right)\vec{k}\right] \tag{4.3}$$

4.2.2 Atmospheric Drag

The drag equation is

$$\vec{F}_{drag} = -\frac{1}{2}\rho v_{rel}\left(\frac{C_d A}{m}\right)\vec{v_{rel}} \tag{4.4}$$

C_d, A, and m correspond to the drag coefficient, ram surface area, and mass of the satellite, respectively. All three of these values vary with attitude and fuel consumption. This program does not consider satellite attitude and assumes no change in mass. As a result, the most challenging part of (4.4) is identifying the correct values for ρ and v_{rel} at every integration step.

Typically, v_{rel} is assumed to be the satellite's velocity with respect to the Earth's, however, the atmosphere does rotate somewhat with the Earth. This consideration is hard to model and will be ignored.

Density is the most difficult factor to correctly estimate. The simplest approximation of the Earth's atmosphere is

$$\rho = \rho_0 \, e^{-\frac{(H-R)}{H_0}} \tag{4.5}$$

where $\rho_0 = 1.3 km/m^3$ and $H_0 = 8.5 km$ [10].

This assumes a constant exponential decay in density, which would be true if the atmosphere was homogenous. However, since air is comprised of multiple species, this model can be inaccurate especially at higher altitudes. Additionally, the Earth's atmosphere varies across multiple time scales including the day/night cycle as well as the Sun's 11-year cycle. This description fails to take these factors into account. One of my main hesitations with Poliastro was this limited description of the atmosphere. However, after I completed the propagator section of the code, Poliastro updated to include the following models: COESA62 and COESA76. These values do not take into account diurnal heating or the solar cycle, but prove to be more accurate than an exponential model

Before discussing alternative models, it is worth noting that each have significant tradeoffs. Highly accurate models will significantly slowdown the execution time of the integrator, especially with high orders. Additionally, models are only valid over a certain altitude range. This project aims to support orbit determination of satellites across a wide range of altitudes. Adding one model would be insufficient. Optimal results would require implementing multiple models with clear communication to the user about their strong suits as well as ensuring they are not used outside of their acceptable range.

Montenbruck and Gill [7] discuss multiple alternative models, including mostly the Jacchia family, which do consider diurnal heating and the solar cycle. It is noteworthy that in order to meaningfully consider the impacts of the solar cycle, we must use recorded or assume solar and geomagnetic indices. For future calculations, it means the introduction of uncertainty that is impossible to quantify until the actual values are measured. For past calculations, it means referencing a large table and interpolating between entries which can be expensive.

4.2.3 Third Body

Third body perturbations is the catch all for non-Earth gravitational factors. Technically this includes the influence of all mass distributed in our solar system and beyond but is often reduced to lunar and solar gravity, although 4.1 also mentioned Venusian and Jovian forces. Together this would give the following equation

$$\vec{F}_{TB} = \vec{F}_{☾} + \vec{F}_{\odot}$$

where

$$\vec{F}_m = \mu_m \left(\frac{\vec{r}_{m/s}}{r_{m/s}^3} - \frac{\vec{r}_m}{r_m^3} \right) \quad m = ☾, \odot \tag{4.6}$$

Here $r_{m/s}$ is the position of the third body with respect the satellite, while r_m is the position of the third body with respect to the Earth.

For the sun and moon, the orbits and gravitational values, $\mu_{☾}$ and $\mu_{\odot}$, are well known. However, how to evaluate the positions of Moon and the Sun with respect to the Earth is a design decision. It can be done a number of ways: integrate the trajectory of all objects, calculate the position of all orbital bodies using their orbital elements, or approximating their location using chebyshev polynomails. Since the gravitational effects of the sun and moon are so small compared to that of the Earth, their exact position does not have to be known. While integrating their position over time can be much more accurate than evaluating using orbital elements, it is not worth the cost in time. Using orbital elements gives an error of .1-1% and can be done quickly with little effort. Chebyshev polynomials provide a middle ground between computation time and high accuracy.

Poliastro handles this by utilizing the astropy function build_ephem_interpolant. This creates a callable object that can be referenced at every interpolation step. These objects are valid between 1800-2050 and are comparable to JPL ephemerides.

4.2.4 Solar Radiation Pressure

Solar radiation pressure is the term given to the change in momentum as a result of incoming light over the area presented to the sun. This area is often different than the ram surface area used to describe drag. While light possesses no mass, it does posses momentum. As light reflects or is absorbed by the satellite, so is the momentum. The equation that describes solar radiation pressure is well known.

$$\vec{F}_{SRP} = -\nu \frac{S}{c} \left(\frac{C_r A}{m} \right) \frac{\vec{r}_{\odot/s}}{r_{\odot/s}} \tag{4.7}$$

ν is the shadow function, which acts as a delta function, expressing whether or not the satellite is in sight of the sun eclipsed by the Earth. S is the solar constant ($1367[W/m^2]$). C_r mimics the coefficient of drag and similarly represents the satellites ability to interact with light. A is is the surface and is often tied to C_r as a single product.

Momentum is transferred differently depending upon whether the incident photon is absorbed or reflected. Additionally, most surfaces have an absorptivity, describing the percentage of photons absorbed. These factors together make the C_r value. As previously discussed in the drag section, attitude is not considered. As a result, C_r and A are assumed to be constants, which is only true for solar-pointing satellites.

Earth radiation pressure, or Albedo, is the exact same physical process, where light is reflecting from the Earth instead of the Sub. The reflectivity of the surface of the Earth various strongly with respect to latitude. Some surfaces such as water and ice reflect significantly. To include albedo as a perturbation, having an accurate model would require converting the satellite's trajectory into Earth-Centered Earth-Fixed (ECEF) frame to get its position above the Earth as well as considering the relative positions of the Sun, Earth, and satellite.

4.3 Poliastro

Poliastro is a clean, easy-to-read library that has proven invaluable to this project. It features a Cowell's method propagator and supports all of the perturbations listed above. It is uses the SciPy DOPRI8 wrapper that calls the original FORTRAN method, vital for saving on computational costs. Although propagating states is still the most significant source of execution time in the program.

In addition to using the FORTAN integrator, Poliastro also takes advantage of Numba's just-in-time decorator allowing for optimization. The decorator flags the method to the compiler for optimization without changing functionality. This is only possible for a few perturbations: J2, J3, exponential drag. The rest involve calling objects or models and are not compatible with the @jit decorator.

Poliastro has also laid the groundwork for external arbitrary perturbations. The Cowell method propagator directly accepts a keyword argument a_d, allowing a user to pass their own function in. This allows us to write improvements, such as to the atmospheric drag or spherical harmonics function, as well as passing a combination of their pre-written perturbations, or even new-to-Poliastro perturbatons such as dynamic solid tide and albedo.

The state_propagator file, serves as a wrapper to poliastro, converting between a state vector and parameters relevant to propagation and Poliastro's orbit object. Additionally, a custom a_d function is written and passed accepting a specific list of perturbations defined by the user of this project, allowing for the user to determine where they stand in the trade-off between time and accuracy.

My main concern with Poliastro is the lack of proper interfacing with TLEs. A third-party library was implemented and expanded to create an Orbit object from a TLE. However, some information is lost such as the parameters in the first line relevant to atmospheric drag. The above description for drag differs from the internal workings of SGP4, the model TLEs were made for. Depending upon the accuracy required, TLEs are a potentially poor way of storing orbital data. At the described epoch, TLEs are associated with roughly a km of error and grow 1-3 km per day [15] [14]. For high precision projects, an alternative means of describing orbits might be required.

4.4 Verification of two-body scenario

To get a baseline for Poliastro's accuracy using the DOPRI8 integrator, I propagated a satellite using two different techniques: the integrator and the Lagrange-Gibbs method. In both cases, the satellite was on a circular orbit $66,666\ km$ from the center of the Earth. No perturbing forces were included, therefore mass not required. Additionally, the exact same value of $\mu_{\overset{+}{\bigcirc}}$ was used, 398600.4418 $[km^2/s^2]$.

The Lagrange-Gibbs solution to the two-body problem is semi-analytical. It solves for the position of a satellite algebraically, without integration. As a result, it does not accumulate error over time. It's error can be reduced to machine precision, $1e-16$. It starts with two equations, clearly related to (4.1).

$$\ddot{F} = -\frac{\mu}{r^3}F \quad \text{and} \quad \ddot{G} = -\frac{\mu}{r^3}G$$

where $F_0 = 1$, $\dot{F}_0 = 0$, $G_0 = 0$, and $\dot{G}_0 = 1$.

Full derivations can be found easily online or in most orbital mechanics textbooks. Skipping steps, we conclude that

$$F(t) = 1 - \frac{a}{r_0}(1 - \cos E)$$

$$G(t) = t + \sqrt{\frac{a^3}{\mu}}(\sin E - E)$$

and

$$\vec{r}(t) = F(t)\vec{r}_0 + G(t)\vec{v}_0$$

Here E is the eccentric anomaly, the orbital distance from periapsis with reference to the

center of the Orbit. To find E, we must first establish the mean anomaly (M), which represents the fraction of a period the object has passed since periapsis. As a result,

$$M = n\ t \quad \text{where} \quad n = \frac{1}{2\ \pi}\sqrt{\frac{\mu}{a^3}} = \frac{1}{T}$$

where T is the period. E is defined with respect to M.

$$M = E - \sin E$$

For a given value of M is it impossible to directly find E. As a result, we employ the Newton-Raphson method. Here we iteratively calculate E until it the update is on the scale of machine precision.

$$E_{k+1} = E_k + \frac{M - E_k + e \sin E_k}{1 - e \cos E_k}$$

With the loop closed on this solution and the integration method being self-explanatory we may now compare these two methods.

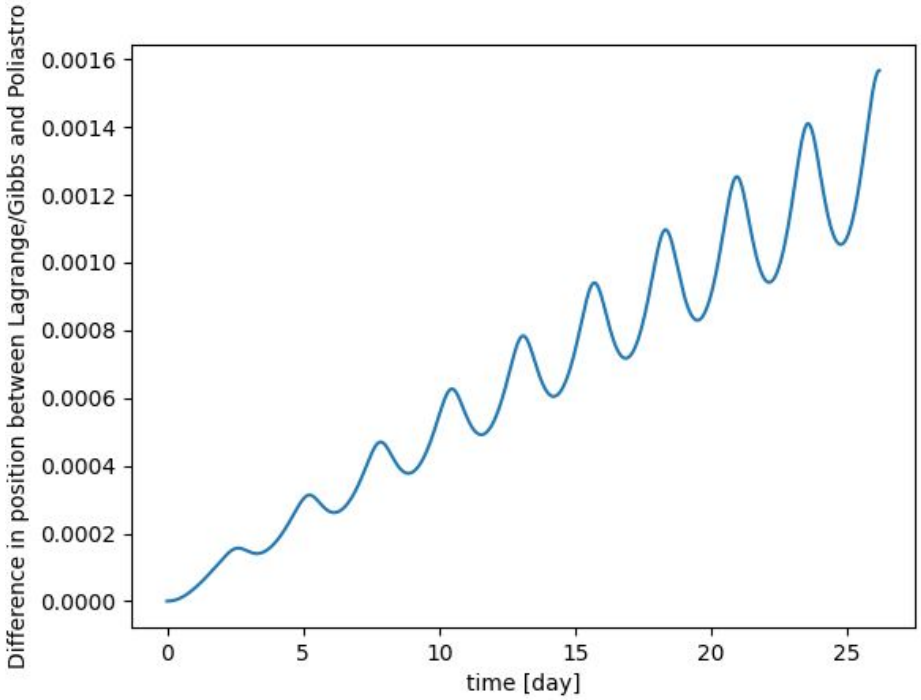

Figure 4.3: Error associated with numerical integration during orbit propagation

4.3 seemingly resembles a line with an exponentially rising oscillation. The exponential nature is much more concerning with respect to long-term error. However, for this example after 25 days, the error is roughly 1.6 *mm*. In practice, if a satellite is lost for more than 7 days, it can be particularly difficult to recover. Propagating for long period shouldn't be a regular occurrence for our users. In comparison, the error associated with storing our states in TLEs far outweighs this source of error.

Chapter 5

Observation Function

The observation function is the bridge between states and the least squares filter. It can take any form provided it resembles a working sensor. In the case of this project, we have assumed all observations will be done via telescope. As a result, this chapter will be heavily reliant upon astronomy concepts. Alternatively, if a user intended to use a RADAR system, this is the only section of the project that would have to be altered.

A distant object at an unknown range can be described by two angles. While it is easier to imagine local angles, azimuth and elevation, astronomers typically utilize right ascension (α) and declination (δ). These angles correspond to locations on the celestial sphere, infinitely far away. For incredibly distant objects such as quasars or galaxies, these objects don't move. For some closer objects, there is some apparent motion. This is particularly true of objects in our own solar system, such as comets or satellites. Before explicitly defining α and δ, we must first discuss the frames in which our motion and observation will occur. In the next subsection, we will describe three.

5.1 Frames

Satellites orbit in a single plane and move inertially with the gravitationally dominant body. Additionally, this motion occurs nearly entirely independent of the main body's rotation. As a result, satellite motion is often described in the Earth-Centered Inertial (ECI) frame. The ECI frame is defined where the x-axis represents direction of the sun in space on the vernal equinox. The z-axis is aligned along the North Pole and the y-axis closes the right-handed system. As the Earth revolves around the Sun, the x-axis continues to point in the same direction. The ECI frame is realized as the GCRS frame by the International Earth Rotation and Reference Systems Service (IERS) [11].

Since the ECI frame does not consider the rotational motion of the Earth, an observer's

location fixed on the surface changes with time. To address this, the Earth-Centered Earth-Fixed frame was defined to aid in satellite observations. Latitude, longitude, and altitude is a pseudo-spherical representation of the ECEF frame. Here, pseudo-spherical is a comment on the non-uniform radius of the Earth. In the ECEF frame, the x-axis is the along the line from the center of the Earth to 0° latitude and longitude. The z-axis is also defined by the North Pole. The y-axis closes the right-handed system and corresponds to the line from the center of the Earth to surface at 0° lattitude and 90° East longitude. The ECI frame is realized by IERS as the International Terrestrial Reference Frame (ITRS). It is geocentric and ensures a no-net rotation with respect to the tectonic motion, and its orientation was given by the BIH orientation at 1984.0.

The third and final frame to be discussed is the International Celestial Reference Frame (ICRF) which is used to define the celestial sphere. While the other two frames are required for any observation function, ICRF is specific to astronomy. Unlike ECI and ECEF, ICRF is not Earth-Centered. Instead, its center is located at the Barycenter of the Solar System. Much like the ECI frame, ICRF is defined by the vernal equinox and the North Pole when the Earth was at J2000. ICRF has been realized multiple times as ICRS, ICRF, and ICRF2, where each of these implementations are defined by separate and increasingly larger numbers of extra-galactic radio sources. ICRF2 is defined by 3414 sources.

Astropy supports all three of these frames with build in transformations between them. These will be utilized to determine celestial and local angles.

5.2 Validation of frames

The ECI and ECEF frames share the same origin, with the same scale, and merely differ by a constant rotation. The period of this rotation was verified analyzing the ground track of a geostationary satellite. The trajectory was determined by propagating forward in time for one period with 100 steps. Each step converted to ECEF, then to LLA. The period was found to be 86164.1s, the number of says in a sidereal day.

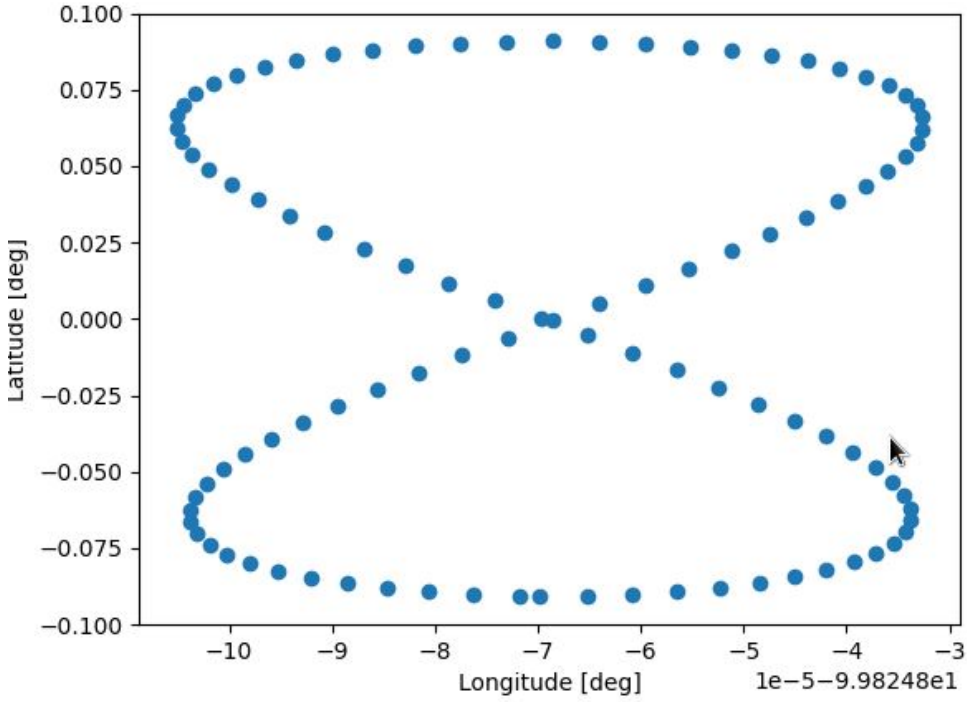

Figure 5.1: Ground track of geostationary satellite originating at 0 degrees latitude and longitude.

To generate this plot, multiple calculations were made resulting in roundoff error. These small deviations have led to a perturbation from a truly geostationary orbit, resulting in the shape above. In terms of deviation in lattitude, we see a $\pm.1$ degrees. With respect to longitude, we see a $\pm 4e-5$ degree deviation. This plot exactly matches the expected ground track of a geostationary satellite, validating the period of rotation is one sidereal day.

In addition to veriyfing the rotational nature of the ECI to ECEF frame conversion, I also compared the norms before and after. There is no difference in scaling, and they share the same origin, as expected. In fact, an example in [15] was validated. Given a state vector in the ECI frame and an observer's latitude and longitude, I was able to verify the range and azimuth and elevation angles within a half km and half degree respectively. The small discrepancy can be attributed to separate propagators or a slight difference in the state vectors.

When attempting to verify the accuracy of the ICRS frame in Astropy, I faced serious issues. I compared my own predictions form right ascension and declination with those of JPL Horizons for multiple satellites. I found that the two sources of predicted observation values differed dramatically. Investigating further, I found that Horizons was less accurate than I was with respect to the Vallado example. Here, I would like to mention that Horizons defines

their angles in the ICRF frame, which may not exactly match the ICRS frame. Conceptually they are are the same and should not be differing as much as we see. When comparing the two, I found that the location of the Earth was off by about 1%, which corresponds to 10,000 km difference. While this error is small with respect to the distance between objects within the solar system, it plays an overwhelming role. When predicting the location of Venus in the sky, the two frames agreed to 1/100 of a degree.

Abstracting the problem, the ECI and ICRS/ICRF frames are similar, in that both are inertial. Despite sharing separate origins, they can define the same celestial sphere. The main difference is that ICRS/ICRF corresponds to J2000 coordinates, while ECI utilizes the equator of the Earth at the current epoch.Physically this corresponds to the precession and nutation of the Earth. Since we have found the ICRS frame to be unreliable with respect to Horizons, we will instead use the ECI frame to define right ascension and declination as it was verified by the Vallado example.

5.3 Defining our angles

As discussed previously, there are typically two sets of angles, celestial and local. While this project strongly favors celestial angles, there are support functions for local angles. This support allows the user to have a prediction of when an object would be visible. However, the observation function returns exclusively celestial angles, which requires all of the input data to also be celestial angles. Calculating both will be discussed in this section. The GUI only supports celestial angles, however, the groundwork has been laid to easy expand to support local angles.

5.3.1 Celestial angles

Both sets of angles are incredibly similar conceptually; one angle measures the vertical angular distance while the other handles the horizontal. They are different in implementation as they occur in two different frames. The ECI frame tracks the objects movement with respect to the stars, independent of the Earth's rotation. Declination measures the angular distance from the North Pole vertically while right ascension measures the angular distance around the celestial sphere. See their definitions below.

$$\alpha(\vec{rr}) = \tan^{-1} \frac{rr_y}{rr_x} \tag{5.1}$$

$$\delta(\vec{rr}) = 90 - \cos^{-1} \frac{rr_z}{|(rr)|} \tag{5.2}$$

The last hurdle involves defining $\vec{rr}$. For objects far away, their position in the celestial sphere is relatively independent of the observer's location. Consider a comet approaching aphelion. Since the object is significantly further away than the relative distance between the observers, both observers will record nearly the same angles. This is the same assumption that allows the definition of the ITRS frame through extragalactic objects. For much closer objects such as satellites this could not be further from the truth. Observers could be thousands of kilometers away observing a satellite a few hundred kilometers overhead.

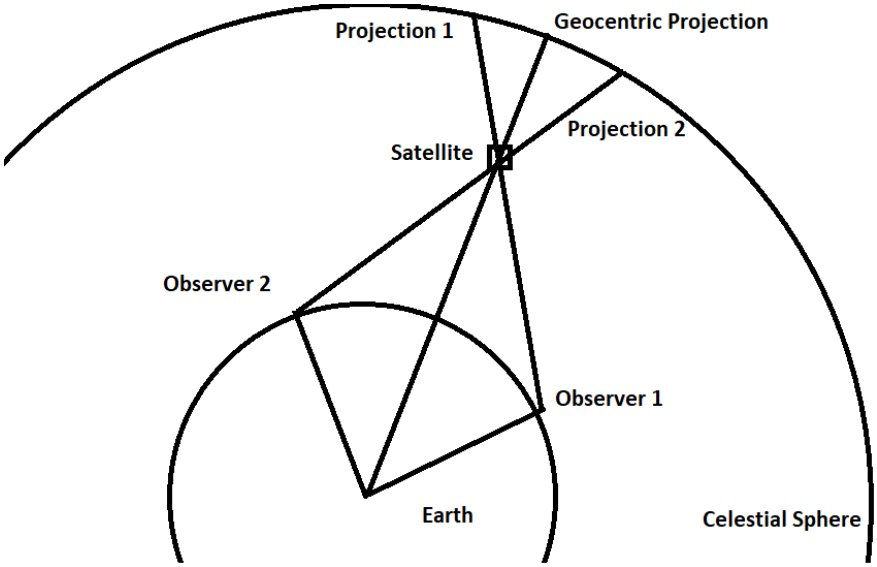

Figure 5.2: Projection of a satellite to the celestial sphere depending upon observer's location

Notably, these projections also differ from a geocentric projection. Taking advantage of the celestial sphere being at infinity, we can use the difference vector (from observer to object) and calculate the two celestial angles. The origin of the ECI frame is the center of the Earth, and since the distance from the center of the Earth to the observer is inherently much less than infinity, this is a safe assumption.

$$\vec{rr} = \vec{r}_{object} - \vec{r}_{observer} \tag{5.3}$$

Vector subtraction must take place in one frame and unless the observer is also a satellite, this requires a transformation. I found the transformation process to be a little slow. Time was saved in execution by converting all of the observer locations to the ECI frame before running the least squares filter. This is directly tied to the previous discussion of execution time in the partial derivative matrix.

5.3.2 Local Angles

The primary difference between the local and celestial angles is the vector $\vec{rr}$ and the conversion from declination to elevation. In the case of pure vertical displacement, or 90° elevation, an object would be along the line from the center of the Earth to the observer's location. This only aligns with the native z-axis of the ECEF and ECI frames when the observer is on the North Pole. As such, we must perform a simple rotational transformation that converts the ECEF frame into our desired local one. This is done in two steps: 1) rotate around the y-axis to bring the z-axis down to the observer's latitude 2) rotate around the z-axis to align the x-axis with the oberserver's longitude.

$$\boldsymbol{R}_y(lat) = \begin{bmatrix} -\sin(lat) & 0 & \cos(lat) \\ 0 & 1 & 0 \\ -\cos(lat) & 0 & -\sin(lat) \end{bmatrix}$$

$$\boldsymbol{R}_z(lon) = \begin{bmatrix} \cos(lon) & -\sin(lon) & 0 \\ \sin(lon) & \cos(lon) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Combining these two matrices and the difference vector in the ECEF frame, we see

$$\vec{rr}_{local} = \boldsymbol{R}_z \boldsymbol{R}_y \ \vec{rr}_{ECEF}$$

Lastly, we see

$$az(\vec{rr}) = 90 - \tan^{-1} \frac{rr_y}{rr_x} \tag{5.4}$$

$$el(\vec{rr}) = \cos^{-1} \frac{rr_z}{|(rr)|} \tag{5.5}$$

5.3.3 Astrometric considerations

Astronomy is an incredibly complex field. This project has opened my eyes to the many considerations required to accurately describe the location of an object in the sky. For starters, the user must be able to calculate the precession and nutation of the Earth since the J2000 epoch. Then the user must be able to account for atmospheric conditions, primarily dependent upon the elevation angle of the target on the sky. Additionally, there is also a light lag where the user must point slightly in front of the desired satellite. Often, these issues are handled within a telescope's software. To accurately describe and summarize these phenomena is outside of the scope of this project, but would be within the purview of a potential user.

Chapter 6

GUI

There are a great many libraries to build a GUI within Python. After assessing the options, I went with Kivy as it was easy to prototype with and aesthetically pleasing out of the box. Most importantly, it was functional on Windows, and it as possible to freeze this code into an executable file.

6.1 Kivy

Building a Kivy product involves working in two environments: *kv* and python. The most significant challenge was understanding the role of each environment. There is significant potential for overlap, as a lot of functionality can be implemented in both environments. To simplify this, I defined a few guidelines for myself. I utilized the *kv* environment to define which physical objects would exist within the GUI as well as their properties such as text size, hints, position, and size of the object. To bridge the gap, I used the *kv* environment to call functions in python. Any changing after initialization of properties was handled in the python environment. This ensures that all of the business logic remained in python and debugging did not require jumping back and forth.

This kivy project takes advantage of the ScreenManager class, allowing delegation of tasks to specific pages. Each of the screens was built as a custom implementation of the base Screen class to fit a unique purpose. Each screen features specific objects at specific locations such as labels or buttons, with little overlap between screens. This design choice allowed me to delegate clusters of related activities, such as dealing with observations, to their own custom screen, decluttering the main screen. Each time the app transitions from one edge to the opposite, the screen changes.

6.1.1 Screens

This project features five main screens, each with an outlined purpose. First, I will describe their role and then feature a screenshot. The first and aptly named MainWindow served to summarize the user's inputs and eventually allow them to run the batch least squares filter. Branching off the MainWindow, there are three screens, allowing the user to edit designated clusters of inputs, and one to run the filter.

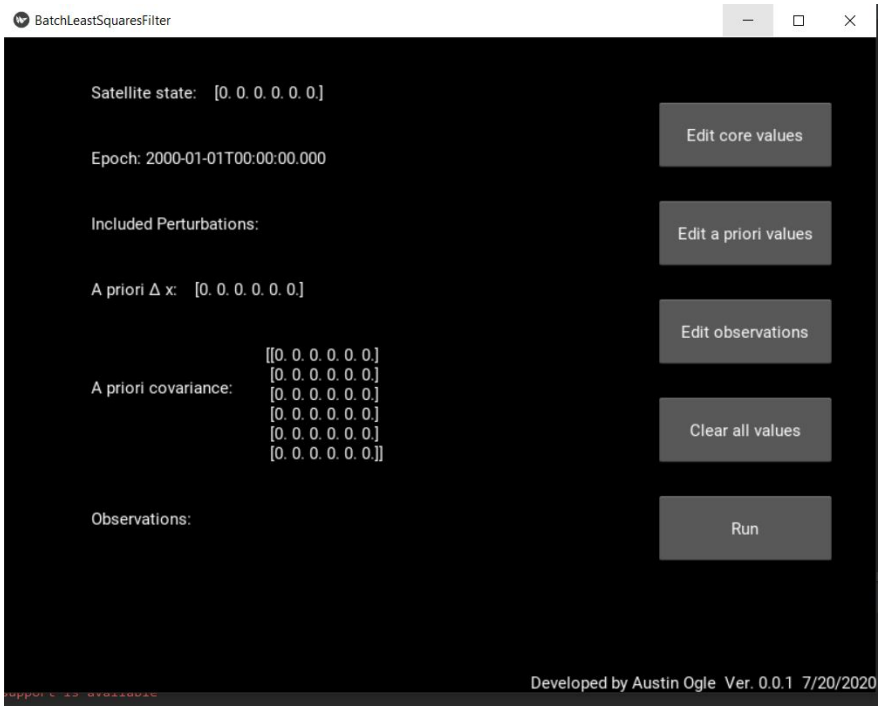

Figure 6.1: Main page. Visible upon first opening the program

The first is the AddCoreValues screen which allows the user to edit the initial state, epoch of the state description, which perturbations should be included in orbit propagation as well as an optional tle input which overrides the state and epoch inputs. For user's that have tle descriptions of the satellite in question, this proves incredibly easy to work with. Interfacing with tles required careful consideration and will be discusses later. The state input expects a 6-element vector corresponding to the position $[km]$ and velocity $[km/s]$ in the ECI (GCRS) frame. The epoch inputs accepts either standard Julien Day (JD) or ISOT format (YYYY-MM-DDTHH:MM:SS.SSS). There is a radio button on the right of the input that allows the user to indicate in which format the input is provided. Notably, the text hint indicating the format is depending upon the state of the radio button. There is a checkbox that allows the user to indicate if they intend to use a TLE description. If checked, the textinput allows the user to input text, changing the background color, hint text, and hint color to indicate the box is available.

This screen also features a button on the bottom indicating the user would like to update their inputs and return to the main page. The inputs on this page are mandatory and as a result there is no option to clear the results or return without making changes. If the user were to give inputs, go to the main page and return, their original inputs will still be in the respective input fields. Before updating or changing pages, each input is checked to see if fits the expected format. If it does not meet expectations, then a popup in created indicating which field(s) are not acceptable. In the case of two incorrect inputs, two popups will be created.

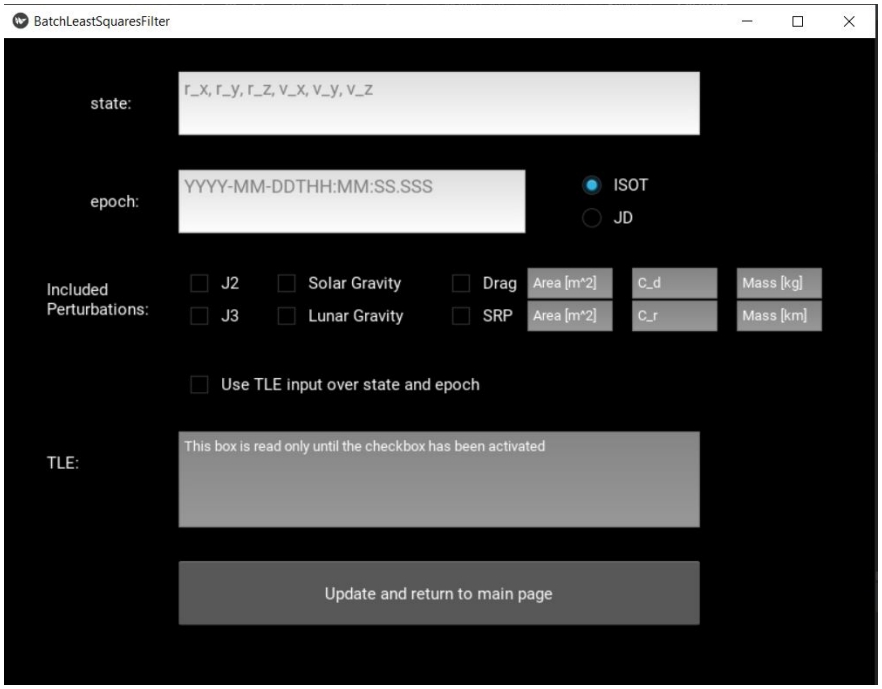

Figure 6.2: AddCoreValues page

The next screen is intended to add *a priori* information. This includes the previous update to the state vector ($\vec{x}_{apr}$) and the corresponding covariance matrix ($\boldsymbol{P}_{apr}$). The first input expects 6 floats separated by commas. The matrix input is similar, except it expects 6 lines of the same style where the delimiter to indicate a new line is the new line character "/n". Since these are non-mandatory inputs, they can be cleared via the red text button on the bottom left. The last element on the page includes is the update and return to main page button. Once again, before moving to the main page, the specific inputs are tested to ensure they are reasonable.

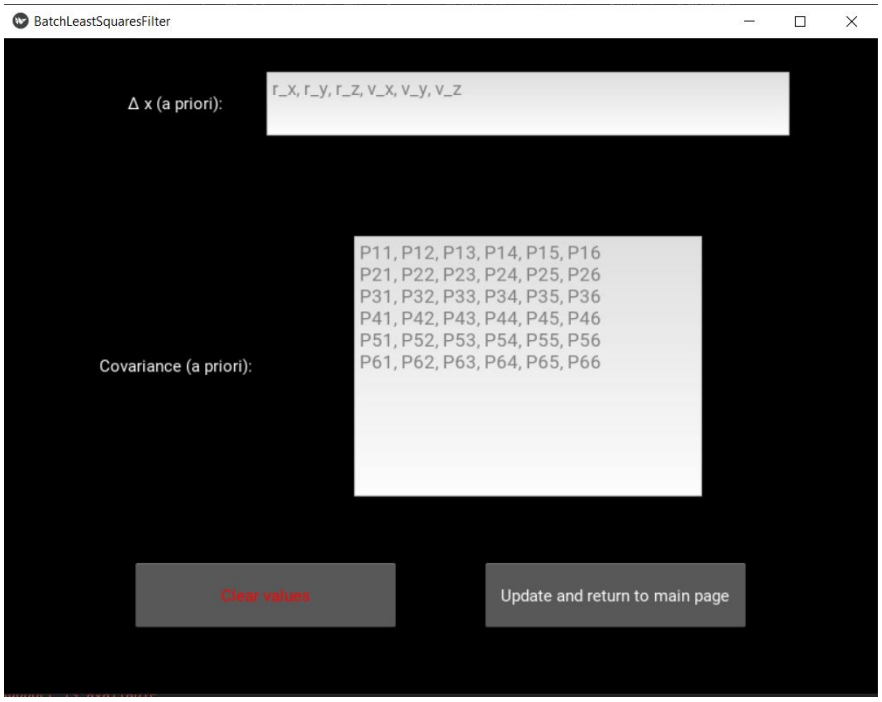

Figure 6.3: AddAPriori page

The last of the input screens is the AddObsScreen, which facilitates editing the users' observations. Each observation includes: the observer's position (lat, lon, alt), the epoch of the observation, the measured right ascension (α) and declination (δ) as well as their corresponding uncertainties. The lat and lon elements can be provided in degree minute seconds format or decimal degrees, indicated by another radio button. The altitude input assumes kilometers for units, indicated by the hint. The epoch is handled the same as above. The measurements and uncertainties assume a float input. Unlike the previous screens, this screen can accept any number of observations. As a result, there is a "add observation" button that captures the information in the fields and builds an observation object. This in turn, clears all of the entries setting the stage for the user to add more observations. The bottom of the page includes a scrollable label which displays the status of the observations stored in memory. In addition to the add observation button, there exists two buttons that round out the functionality. The first clears all of the observations stored in memorry and the seconds returns the user to the main page. A more nunanced method of editing and deleting previously input observations has been identified as an item for future work. This would feature some sort of plot which gives the squared sum of the measurements on the y-axis with the time of the observations along the x-axis. The user could highlight cluster's of data and indicate whether to delete, mute, or only consider the selected points.

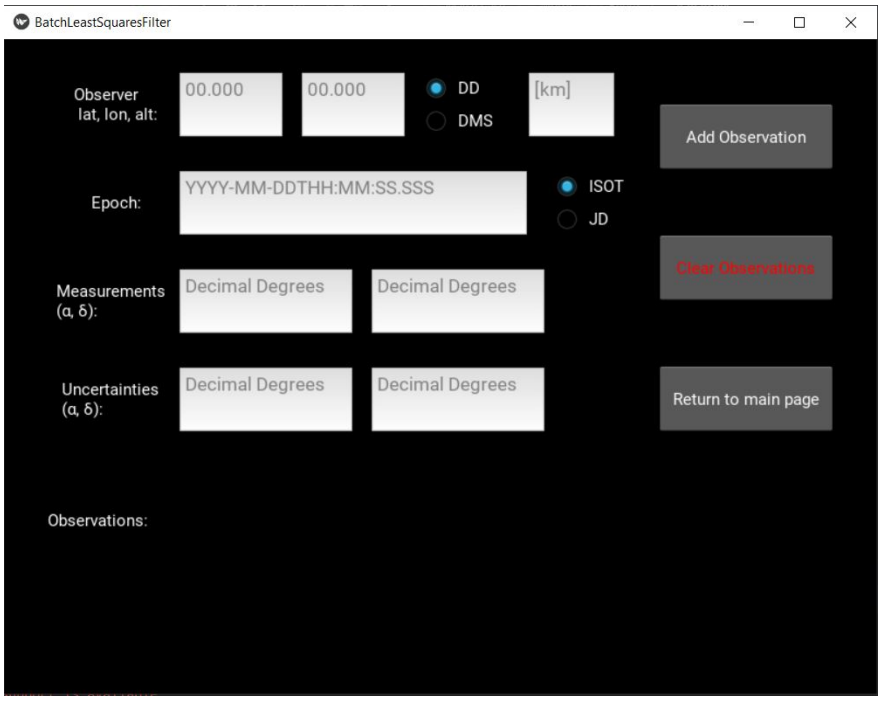

Figure 6.4: AddObsScreen page

The final screen features the output of the batch least squares filter. Internally, this is handled as a single object with a tostring() class function. This object includes the initial state guess, the corresponding epoch, the resulting state from the filter, the update to the state to get the result, and the corresponding covariance matrix. Except for the epoch, each of these attributes are NumPy arrays. To preserve accuracy, they are printed with 16 decimal places.

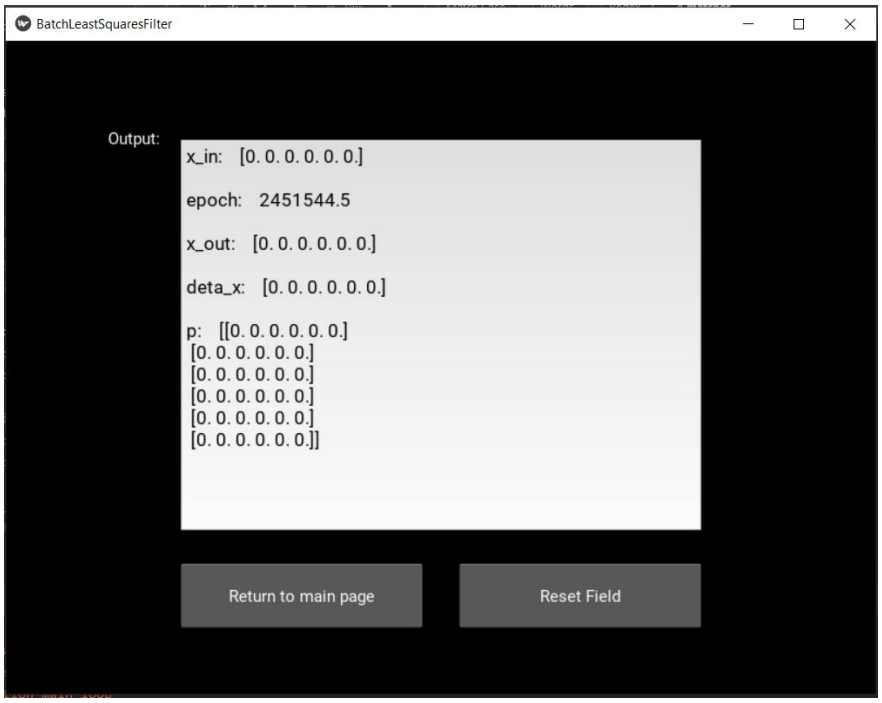

Figure 6.5: Filter results page

TLE

In Python, I have borrowed and expanded the TLE object from the tle-tools library available for Linux, notably not available for windows developers. This class was built in an object-oriented style separate from the main body of code. If I had come across this class earlier, I would have refactored section of the code to make it easier to read. For example, similar to Poliastro's Orbit object, I would have implemented a State object that featured a state, epoch, and covariance matrix as attributes with propagate as a class method.

The TLE object features a great many attributes, each corresponding to an element of the tle. This includes the name, satellite catalog number, classification, international designator, epoch year, epoch day, first derivative of the mean motion divided by 2, second derivative of the mean motion divided by 6, B star, the set number, inclination, right ascension of the ascending node, eccentricity, argument of periapsis, mean anomaly, mean motion, and number of full revolutions. In addition to these attributes, the TLE object also had the following properties: semi-major axis, epoch, true anomaly, and period. The B star, first and second derivatives are values directly tied to the SGP4 propagator and consequently held no meaning to this project. They were interpreted as strings. The B star values possess some meaning about the effect of aerodynamic drag.

$$B^* = \frac{\rho_0 C_d A}{2m}$$

While theoretically we could have divided out the reference density and included this directly into a atmospheric drag perturbation function, this would have involved writing a new perturbation function to be passed to Poliastro. While this is possible, it was not identified as a priority and could be included in the future. It is still possible to include atmospheric drag as a perturbation, this only requires providing C_d, A, and m in addition.

The TLE object features five class methods. The first acts as the only constructor from_lines(). This function accepts a TLE and builds the object. The next class function update() accepts a state and epoch as inputs and will change the relevant attributes. This allows us to modify the original tle description with the result of the filter. The next function named to_string() is self expanatory. This outputs the tle object per the NORAD standard. The last two functions are similar to_orbit() and to_state(). Both convert the tle into useable objects for specific purposes.

In the original tle-tools library, the TLE object featured the from_lines(), to_orbit(). The other three were added to meet the requirements of my project. The to_string() function was the most difficult to accurately produce as there were multiple inconsistencies in the tle_string to TLE object loop. When debugging this process, I found significant issues in the original code where values such as the epoch were being rounded. The tle_string to TLE loop is without error and has been tested with a wide variety of orbits. The update() function proved to be more complex than originally anticipated. The first issue was the NORAD standard of angles between 0-360 degree, where our math gives results from -180 to 180 degrees. Additionally, counting the number of revolutions between epochs was handled with care. The mean anomaly was directly considered during this process.

6.1.2 Layouts

The layout is the first required layer to the Screen Object. Layouts come in multiple forms and are responsible for how GUI elements are placed. The options include grids, boxes, pages, and floating layouts. I went with the FloatLayout as it allowed me to position elements directly where I wanted them. This layout provided the most flexibility.

6.1.3 Objects

The most common object within this GUI is the Label. Labels exist to display text. The color, size, and font can all be selected in either environment.

Much like to Labels, TextInputs have basic attributes such as colors and font sizes and so on. In addition, TextInputs can be read only, allow multiple lines, and even trigger events, for

example the user hitting the enter key. They allow basic shortcuts such as copy and paste. The hint_text attribute of the TextInputs was used regularly to give the acceptable format of entries. Once the user adds a character to the TextInput, the hint disappears.

Buttons were used to move between screens or trigger functions that then utilized the entries of the TextInputs. If a button deleted data, the text was red to indicate caution.

Checkboxes were used as True/False values. This helped determine the user's intent. The cases of this included selecting a desired time format, latitude and longitude format, as well as whether or not the user planned on inputting a tle.

Chapter 7

Testing

It is difficult to identify a pass/fail scenario for a least squares filter. The outputs vary depending upon several design decisions, which theoretically should result in the same answer. Both numerical integration and differentiation are present in this project, which depend upon user-defined value independent of the problem itself. As such, the specific output depends upon a number of factors and is impossible to nail down as definitively correct. Consequently, a subjective approach has been taken to evaluate the accuracy of the project.

All tests were conducted with simulated data. Real data with a telescope was limited due to a number of reasons. Most significantly, I am writing this in July of 2020 as the second wave of COVID-19 is surging in Florida. Consequently, and for good reason, there exist significant limitations on department's telescopes. While my advisor does have a suitable telescope at home, we were unable to develop sufficient possible sightings and follow through with enough sightings to conduct a meaningful test. We have had particularly humid nights and even a Saharan dust storm.

The system-level tests conducted on this product were all based on the same concept. Initially there are two states: $\vec{x}_{true}$ and $\vec{x}_{guess}$. All observations were based on the true state vector while the guess was provided to the LSQ filter. In theory, the filter should exactly converge to the true state vector. For some scenarios, gaussian noise was added to the observed values. This should have the effect of perturbing the converged result, however, it should be in the proximity of the true orbit. As these tests demonstrate, proximity to the exact solution depends primarily on the noise or error associated with the measurements.

Four scenarios were tested: LEO, HEO at perigee, HEO at apogee, and GEO. These cases should cover the extremes of the expected use cases. The following tables are the direct testing results. Gaussian noise refers to the noise on the measurements and has the units arcseconds. This value was also used as the standard deviation of the measurement itself. The standard deviations are the square root of the diagonal of the covariance matrix. Each scenario features 10 observations predicted at 1/32 of the period incrementally after the

original epoch.

7.1 Testing Scenarios

For the LEO test, the following criteria were used. $\vec{x}_{true} = [6248, 2779, 3543, 4.53, -1.822, -5.626]$. $\vec{x}_{guess} = [5748, 2679, 3443, 4.33, 0-1.922, -5.726]$. This gives an initial offset of $[500, 100, 100, .2, .1, .1]$ km and km/s to $[\vec{rv}]$ respectively.

Table 7.1: Testing convergence results for a sample LEO satellite. See leo.py in src/verification/formal for additional context.

Observation Noise	iterations	$\lvert\vec{x}_{out} - \vec{x}_{true}\rvert$ variances
1e-5'	9	[4.87e-7, 6.68e-8, 1.38e-7, 8.51e-10, 4.53e-10, 1.52e-9] [5.93e-7, 7.74e-7, 6.20e-7, 8.72e-10, 1.11e-9, 1.99e-9]
1'	7	[2.21e-1, 5.66e-1, 1.49e-1, 4.59e-4, 3.20e-4, -.61e-4] [5.93e-1, 7.74e-1, 6.20e-1, 8.73e-4, 1.1e-3, 1.99e-3]
2'	7	[.870, 8.51e-1, 2.964e-1, 1.40e-3, 2.20e-4, 2.34e-3] [1.19, 1.55, 1.24, 1.7e-3, 2.21e-3, 3.97e-3]
5'	7	[1.34e-1, 2.25, 5.64e-1, 1.05e-3, 1.50e-3, 7.12e-3] [2.97, 3.87, 3.10, 4.36e-3, 5.54e-3, 9.93e-3]

For the HEO at perigee test, the following criteria were used. $\vec{x}_{true} = [7100, 100, 100, .2, 10.2, .1]$. $\vec{x}_{guess} = [6600, 0, 0, 0, 10, 0]$. This gives an initial offset of $[500, 100, 100, .2, .2, .1]$.

Table 7.2: Testing convergence results for a sample HEO satellite at perigee. See heo_perigee.py in src/verification/formal for additional context.

Observation Noise	iterations	$\lvert\vec{x}_{out} - \vec{x}_{true}\rvert$ variances
1e-5'	9	[6.03e-6, 7.72e-7, 4.47e-6, 1.06e-8, 5.22e-9, -9.16e-9] [1.04e-5, 7.84e-6, 1.37e-5, 1.88e-8, 1.10e-8, 1.33e-8]
1'	5	[.804, 6.65e-2, .918, 2.27e-3, 5.13e-5, 7.73e-4] [1.04, .784, 1.37, 1.89e-3, 1.10e-3, 1.33e-3]
2'	6	[1.90, .701, .520, 1.34e-3, 1.12e-4, 2.50e-3] [2.09, 1.57, 2.74, 3.77e-3, 2.20e-3, 2.66e-3]
5'	5	[3.34, .240, .341, 9.62e-3, 7.08e-4, 8.68e-3] [5.22, 3.92, 6.85, 9.45e-3, 5.50e-3, 6.64e-3]

For the HEO at apogee test, the following criteria were used. $\vec{x}_{true} = []$. $\vec{x}_{guess} = [32000, 0, 0, 0, 2, 0]$. This gives an initial offset of $[5000, 1000, 1000, .2, .2, .01]$.

Table 7.3: Testing convergence results for a sample HEO satellite at apogee. See heo_apogee.py in src/verification/formal for additional context.

Observation Noise	iterations	$\lvert\vec{x}_{out} - \vec{x}_{true}\rvert$ variances
1e-5'	7	[1.06e-3, 1.99e-5, 5.44e-5, 9.76e-8, 3.15e-8, 6.32e-9]
		[4.21e-3, 7.74e-5, 1.52e-4, 2.99e-7, 1.85e-7, 2.16e-8]
1'	5	[65.4, 8.50, 4.40, 1.55e-3, 5.81e-3, 9.88e-4]
		[416, 7.78, 15.1, 2.98e-2, 1.82e-2, .215e-3]
2'	6	[34.7, 5.66, 13.1, 1.47e-3, 1.26e-3, 2.35e-4]
		[838, 15.5, 30.1, 5.96e-2, 4.68e-2, 4.34e-3]
5'	6	[440, 12.3, 29.2, 2.91e-2, 1.72e-2, 3.98e-3]
		[2000, 38.5, 71.4, .145, 8.73e-2, 1.06e-2]

For the GEO test, the following criteria were used. $\vec{x}_{true} = [-23828.9136, -30695.0020, 1003.5110, 2.5098, -$ $\vec{x}_{guess} = [-27828.9136, -31695.0220, 3.5110, 2.3098, -2.0286, -.0019]$. This gives an initial offset of $[5000, 1000, 1000, .2, .2, .01]$.

Table 7.4: Testing convergence results for a sample GEO satellite. See geo.py in src/verification/formal for additional context.

Observation Noise	iterations	$\lvert\vec{x}_{out} - \vec{x}_{true}\rvert$ variances
1e-5'	9	[2.34e-5, 5.80e-5, 2.25e-5, 6.93e-9, -6.60e-9, .5,05e-9]
		[1.79e-3, 3.02e-3, 6.09e-5, 2.08e-8, 3.56e-8, 4.85e-9]
1'	7	[7.13, .180, -1.97, 1.35e-3, 2.18e-3, -5.54e-4]
		[18.0, 30.2, 6.09, 2.88e-3, 3.56e-3, 4.84e-4]
2'	7	[18.5, 3.75, 4.17, 1.898e-4, 7.58e-4, 1.43e-3]
		[35.9, 60.3, 12.2, 4.16e-3, 7.12e-3, 9.70e-4]
5'	7	[49.9, 25.4, 17.5, 2.18e-3, 3.25e-3, 2.56e-3]
		[89.5, 150, 30.4, 1.04e-2, 1.78e-2, 2.43e-3]

7.2 Interpretation

The most interesting lesson learned through these scenarios related to the filter's ability to converge to the "true" orbit. The only strange case was the cross-track error for the HEO orbit at apogee was two orders of magnitude larger than down track. With respect to the other scenarios, everything else gave anticipated results. For the scenarios where the satellite was close to the observer, HEO at perigee and LEO, the solution converged much closer to the truth with smaller uncertainties. I would attribute this to the definition of $\boldsymbol{A}$, where changes in the state would have a larger impact on the predicted observational values than the situations where the satellite was further away.

With respect to covariance, a statement in the Covariance analysis subsection was verified. The uncertainty associated with the converged solution is directly correlated to the noise of the measurements. 7.2 is the clearest demonstration of this possible. The variance in r_x for each of the 4 noise cases was almost exactly equal to the noise.

While not demonstrated in the above data, I found that the distance between the guess and solution was often negligible. For one execution of HEO at perigee, I used a distance of $[5000, 1000, 1000, .2, .2, .01]$ with respect to initial state of $[6600, 0, 0, 0, 10, 0]$. The converged state and variances were the same in magnitude and seemingly only different due to the gaussian nature of the noise. This validates a remaining hypothesis that the LSQ filter has one global minimum that it will always converge to. I have not come across a circumstance where this statement did hold true.

From the above data, it is my conclusion that the LSQ filter is functioning properly. The filter does consistently converge reasonably to the target orbit under various circumstances. Given that there are no mission constraints, large uncertainties are not an issue, provided they are subject to the anticipated parameters. These parameters include the nature of the target orbit, the quality, and the quantity of observations. This report has sufficiently described their interplay such that it would be possible to design a system for situational awareness with actual requirements.

Chapter 8

Future Expansion

This project serves as a clear implementation of a Batch Least Squares Filter, however, with all software development projects, there exist remaining tickets that were never prioritized. These include:

1. Expand the GUI
2. Include interface to STK incase user has license and want to use HPOP
3. Update state vector to be in terms of modified equinoctial orbital elements instead or position and velocity
4. Better integrator RKN12(10)
5. Support additional perturbations (spherical harmonics, Earth tide, relativistic effects, albedo)

The two most obvious expansions for the GUI would consist of more nuanced editing of inputted observations and the ability to visualize the results. While we provide an output with the covariance matrix for clear context, it still may be unclear how this impacts the satellite's mission. Providing a plot of the satellite orbiting the Earth as well as which observations are most helpful would aid the user in mission execution as well as having confidence in the solution.

STK is callable from a python script. Should a user have a license, it may prove to be a more accurate tool than Poliastro and SciPy.

As mentioned in the Least Squares chapter, updating the state vector in terms of modified equinoctial orbital elements has the advantage of being more stable. There is a singularity when the $i = 180°$. It would interesting to compare their impact the current r, v description of the state vector.

As the State Propagation chapter showed, there exists numerical error when propagating over long period of time. While 25 days only added $1.6mm$ of error, a much longer propagation period could have a noticeable impact on the results. In comparison, tles inherently include about $1km$ of error, which grows $1 - 3km$ per day. For particularly high accuracy cases, it might prove useful to have a more accurate, albeit slower integrator.

Lastly, there are several perturbations not included in Poliastro. While the major ones are, the most important to update would be the spherical harmonics description. The current J2 and J3 functions seem too simple for practical use. While they are from [3], other books feature alternatives.

Bibliography

[1] Roger R. Bate, Donald D. Mueller, Jerry E. White, Fundamentals of Astrodynamics, Dover Publications Inc. (1971)

[2] Richard H. Battin, An Introduction to the Mathematics and Methods of Astrodynamics, Revised Edition, AIAA Education Series, American Institute of Aeronautics and Astronautics, Inc. (1999)

[3] Howard D. Curtis, Orbital Mechanics for Engineering Students, Butterworth-Heinemann, 3rd Edition

[4] David Hobbs, Preben Bohn, Precise Orbit Determination for Low Earth Orbit Satellites, Annals of the Matie Curie Fellowship Association, 4, 1-7 (2006)

[5] Jean Kovalesky, Comparison of "Old" and "New" Concepts: Reference Systems, IERS Technical Note, 31-34, (2002)

[6] Adrea Milani, The Asteroid Identification Problem, Icarus 137, 269-292 (1999)

[7] Oliver Montenbruck, Eberhard Gill, Satellite Orbits: Models, Methods, Applications, 4th Printing, Springer-Verlag Berlin Heidelberg (2012)

[8] Rodolpho Vilhena de Moraes, Aurea Aparecida da Silva, and Helio Koiti Kuga, Simple Orbit Determination Using GPS Based on a Least-Squares Algorithm Employing Sequential Givens Rotations, Hindaqi Publishing Corporation, Mathematical Problems in Engineering, 2007, Article Id 49781, 2007

[9] D. H. Oza, T.L. Jones, M. Hakimi, Mina V. Samii, C. E. Doll, G. D. Mistretta, R. C. Hart, TDRSS-User Orbit Determination Using Batch Least-Squares and Sequential Methods, Flight Mechanics Estimation Theory Symposium, 129-144, (1993)

[10] Imke de Pater, Jack J. Lissauer, Planetary Sciences, Cambridge University Press, Second Edition, (2010)

[11] Gerard Petit, Brian Lezum, IERS Conventions (2010), International Earth Rotation and Reference Systems Service (IERS), IERS Technical Note No. 26, Verlag des Bundesamts für Kartographie und Deodäsie, Frankfurt am Main (2010)

[12] Juan Luis Cano Rodriguez, Helge Eichhorn, Frazer McLean, Poliastro: An Astrodynamics Library written in Python with Fortran Performance, 6th International Conference on Astrodynamics Tools and Techniques, Darmsradt, (2016)

[13] Byron D. Tapley, Bob E. Schutz, George H. Born, Statistical Orbit Determination, Elsevier Academic Press (2004)

[14] David A. Vallado, Paul Crawford, SGP4 Orbit Determination, AIAA 2008-6770, (2008)

[15] David A. Vallado, Fundamentals of Astrodynamics and Applications: Fourth Edition, Space Technologies Library, Microcosm Press (2013)

[16] World Geodetic System - 1984 (WGS-84) Manual, Second Edition, International Civil Aviation Organization, (2002)

Chapter 9

Appendix A - Code

All of the code is available publicly visible on my github account. https://github.com/ausogle/astroThes. The repository name may change, but the project will remain public. The code featured below fits into three categories. Src- the main executable code. Verification- code written to verify the tools I built/am using are working correctly and lastly Tests- unittests.

9.1 Src

```python
from typing import List
import numpy as np
from astropy.time import Time
from src.enums import Angles, Frames


class FilterOutput:
    """
    Object intended to store all relevant information from the filter.
    """
    def __init__(self, x_in=np.zeros(6), epoch=Time("2000-01-01T00:00:00.000", format='isot', scale='utc'),
                 x_out=np.zeros(6), delta_x=np.zeros(6), p=np.zeros((6, 6))):
        self.x_in = x_in
        self.epoch = epoch
        self.x_out = x_out
        self.delta_x = delta_x
        self.p = p

    def tostring(self) -> str:
        """
        Converts Filter Output into a string capable of being printed into a file
        """
        output_string = "x_in:\t" + np.array2string(self.x_in, precision=16)
        self.epoch.format = 'jd'
        output_string += "\n\nepoch:\t" + str(self.epoch.value)
        output_string += "\n\nx_out:\t" + np.array2string(self.x_out, precision=16)
        output_string += "\n\ndeta_x:\t" + np.array2string(self.delta_x, precision=16)
        output_string += "\n\np:\t" + np.array2string(self.p, precision=16)
        return output_string


class Observation:
    """
    Object intended to provide all relevant information to the predictor function.
    """
    def __init__(self, position, frame: Frames, epoch_obs:
```

```
36 Time, obs_values: np.ndarray, obs_type: Angles, obs_sigmas=
   np.ones(2)):
37         self.position = position   # If LLA frame, this is
   a list. If ECI/ECEF this is a numpy array. See cleaning.py
38         self.frame = frame
39         self.epoch = epoch_obs
40         self.obs_values = obs_values
41         self.obs_sigmas = obs_sigmas
42         self.obs_type = obs_type
43 
44     def tostring(self):
45         if self.frame == Frames.LLA:
46             return "[" + self.epoch.fits + ", [" +\
47                    str(self.position[0].value) + ", " + str
   (self.position[1].value) + ", " + str(self.position[2].
   value) + \
48                    "], " + str(self.obs_values[0]) + "\
   u00B1" + str(self.obs_sigmas[0]) + ", " + \
49                    str(self.obs_values[1]) + "\u00B1" + str
   (self.obs_sigmas[1]) + "]"
50         else:
51             return "[" + self.epoch.fits + ", [" +\
52                    str(self.position[0]) + ", " + str(self.
   position[1]) + ", " + str(self.position[2]) + \
53                    "], " + str(self.obs_values[0]) + "\
   u00B1" + str(self.obs_sigmas[0]) + ", " + \
54                    str(self.obs_values[1]) + "\u00B1" + str
   (self.obs_sigmas[1]) + "]"
55 
56 
57 
58 class PropParams:
59     """
60     Object intended to provide all relevant information to
   the propagate function
61     """
62     def __init__(self, epoch_i: Time):
63         self.epoch = epoch_i
64         self.perturbations = {}
65 
66     def add_perturbation(self, name, perturbation):
67         self.perturbations[name] = perturbation
68 
69     def tostring(self):
70         output = ""
71         for key, value in self.perturbations.items():
```

```
 72             if output != "":
 73                 output += ", "
 74             output += key.value + ""
 75         return output
 76
 77
 78 class J2:
 79     """
 80     Object intended to include all values required by
    poliastro/core/perturbation.py J2_perturbation()
 81     """
 82     def __init__(self, J2: float, R: float):
 83         self.J2 = J2
 84         self.R = R
 85
 86
 87 class J3:
 88     """
 89     Object intended to include all values required by
    poliastro/core/perturbation.py J3_perturbation()
 90     """
 91     def __init__(self, J3: float, R: float):
 92         self.J3 = J3
 93         self.R = R
 94
 95
 96 class Drag:
 97     """
 98     Object intended to include all values required by
    poliastro/core/perturbation.py atmospheric_drag()
 99     """
100     def __init__(self, R: float, C_D: float, A: float, m:
    float, H0: float, rho0: float):
101         self.R = R
102         self.C_D = C_D
103         self.A = A
104         self.m = m
105         self.H0 = H0
106         self.rho0 = rho0
107
108
109 class ThirdBody:
110     """
111     Object intended to include all values required by
    poliastro/core/perturbation.py Third_Body(). Can be used
    for
```

```
112     Lunar and Solar gravity.
113     """
114     def __init__(self, k_third: float, third_body):
115         self.k_third = k_third
116         self.third_body = third_body        # Build from
    ephemeris
117
118
119 class SRP:
120     """
121     Object intended to include all values require by
    poliastro/core/perturbation.py radiation_pressure().
122     A_over_m is the new implementation. May lead to issues
    .
123     """
124     def __init__(self, R: float, C_R: float, A: float, m:
    float, Wdivc_s: float, star):
125         self.R = R
126         self.C_R = C_R
127         self.A = A
128         self.m = m
129         self.Wdivc_s = Wdivc_s
130         self.star = star                    # Build from
    ephemeris
131
```

```python
import numpy as np
from src.observation_function import y
from src.state_propagator import state_propagate
from src.dto import Observation, PropParams, FilterOutput
from scipy.linalg import solve_banded
from typing import List


def milani(x: np.ndarray, observations: List[Observation], prop_params: PropParams,
           a_priori=FilterOutput(), dr=1, dv=.05, max_iter=15) -> FilterOutput:
    """
    Scheme outlined in Adrea Milani's 1998 paper "Asteroid Idenitification Problem". It is a least-squared psuedo-newton
    approach to improving a objects's orbit description based on differences in object's measurement in the sky versus
    where it was predicted to be.

    :param x: State vector of the satellite at a time separate from the observation
    :param observations: List of observational objects that capture location, time, and direct observational parameters.
    :param prop_params: Propagation parameters, passed directly to propagate()
    :param a_priori: Output from a previous iteration
    :param dr: Spatial resolution to be used in derivative function
    :param dv: Resolution used for velocity in derivative function
    :param max_iter: Maximum number of iterations for Least Squares filter
    :return: A more accurate state vector at the same time as original description, not observation
    """

    n = len(x)
    x_in = x - np.zeros(6)
    delta = np. array([dr, dr, dr, dv, dv, dv])
    delta_x = np.ones(n)
    rms_old = 1e10                  #Following two definitions must break loop for first two iterations
    rms_new = 1e8
```

```
32
33      i = 0
34      while not stopping_criteria(rms_new, rms_old) and i <
    max_iter:
35          c = np.zeros((n, n))
36          d = np.zeros(n)
37          rms_old = rms_new - 0
38          for observation in observations:
39              ypred = y(state_propagate(x, observation.epoch
    , prop_params), observation)
40              yobs = observation.obs_values
41              xi = yobs - ypred
42              w = np.diag(1/np.multiply(observation.
    obs_sigmas, observation.obs_sigmas))
43
44              b = dy_dstate(x, delta, observation,
    prop_params)
45              c += b.T @ w @ b
46              d += b.T @ w @ xi
47          if np.array_equal(a_priori.p, np.zeros((6, 6))):
48              delta_x = get_delta_x(c, d)
49          else:
50              l = get_inverse(a_priori.p)
51              delta_x = get_delta_x(l + c, l @ a_priori.
    delta_x + d)
52          xnew = x + delta_x
53          x = xnew - np.zeros(n)
54          rms_new = np.sqrt((xi.T @ w @ xi)/n)
55          i = i+1
56
57      p = a_priori.p + get_inverse(c)
58      covariance_residual = np.linalg.norm(p @ c - np.eye(6))
59      print("Covariance Residual")
60      print(covariance_residual)
61
62      output = FilterOutput(x_in, prop_params.epoch, x,
    delta_x, p)
63      return output
64
65
66  def direction_isolator(delta: np.ndarray, i: int):
67      """
68      direction_isolator() manipulates the delta array to
    return an empty array with the exc
69
70      :param delta: Input array of step sizes for calculating
```

```
 70  derivatives around the state vector
 71      :param i: the element of delta desired to be preserved
    .
 72      :return: Near empty array, where the ith element is
    the ith element of delta.
 73      """
 74      m = np.zeros((6, 6))
 75      m[i][i] = 1
 76      return m @ delta
 77
 78
 79 def dy_dstate(x: np.ndarray, delta: np.ndarray,
    observation: Observation, prop_params: PropParams, n=2
    ) -> np.ndarray:
 80      """
 81      dy_dstate() calculates derivatives of the prediction
    function per state vector and returns a matrix where each
 82      element is the column corresponds to an element of the
     prediction function output and the row corresponds
 83      to an element being varied in the state vector. Uses a
     second-order centered difference equation.
 84
 85      :param x: State vector
 86      :param delta: variation in position/velocity to be
    used in derivatives
 87      :param observation: Observational parameters, passed
    directly to Ffun0(
 88      :param prop_params: Propagation parameters, passed
    directly to propagate()
 89      :param n: number of elements in prediction function
    output
 90      :return: Matrix of derivatives of the prediction
    function in position/velocity space
 91      """
 92      m = len(x)
 93
 94      a = np.zeros((n, m))
 95      for j in range(0, m):
 96          temp1 = state_propagate(x + direction_isolator(
    delta, j), observation.epoch, prop_params)
 97          temp2 = state_propagate(x - direction_isolator(
    delta, j), observation.epoch, prop_params)
 98          temp3 = (y(temp1, observation) - y(temp2,
    observation)) / (2 * delta[j])
 99          for i in range(0, n):
100              a[i][j] = temp3[i]
```

```
101     return a
102 
103 
104 def get_delta_x(a: np.matrix, b: np.ndarray, upper=5,
    lower=5) -> np.ndarray:
105     """
106     Solves the system of equation using the scipy wrapper
    for LAPACK's dgbsv function.
107     Requires converting a into ab matrix. Notably, for our
     system the upper and lower bandwidths are both 5.
108 
109     :param a: A matrix in normal equation. For our problem
     this is C
110     :param b: b vector in normal equation. For our problem
     this is
111     :param upper: Upper bandwidth of matric C
112     :param lower: Lower bandwidth of matrix c
113     """
114     ab = diagonal_form(a, upper=upper, lower=lower)
115     x = solve_banded((upper, lower), ab, b)
116     residual = a@x-b
117     print("residual")
118     print(np.linalg.norm(residual))
119     return x
120 
121 
122 def get_inverse(c: np.ndarray):
123     """
124     Inverts C matrix using banded solver.
125 
126     :param c: Normal matrix required to invert for
    covariance matrix
127     """
128     inv = get_delta_x(c, np.eye(6))
129     # residual = c @ inv - np.eye(6)
130     # print(np.linalg.norm(residual))
131     return inv
132 
133 
134 def diagonal_form(a: np.matrix, upper=5, lower=5) -> np.
    matrix:
135     """
136     Ripped from github.com/scipy/scipy/issues/8362. User
    Khalilsqu wrote the following function.
137     Converts a into ab given upper and lower bandwidths.
138     Follows notes at people.sc.kuleuven.be/~raf.vanderbril
```

```
138 /homepage/publications/papers_html/fr_lev/node16.html
139
140     :param a: A matrix in normal equation. For our problem
     this is C
141     :param upper: Upper bandwidth of a
142     :param lower: Lower bandwidth of a
143     """
144     n = a.shape[1]
145     ab = np.zeros((2*n-1, n))
146     for i in range(n):
147         ab[i, (n-1)-i:] = np.diagonal(a, (n-1)-i)
148
149     for i in range(n-1):
150         ab[(2*n-2)-i, :i+1] = np.diagonal(a, i-(n-1))
151     mid_row_inx = int(ab.shape[0]/2)
152     upper_rows = [mid_row_inx - i for i in range(1, upper+
    1)]
153     upper_rows.reverse()
154     upper_rows.append(mid_row_inx)
155     lower_rows = [mid_row_inx + i for i in range(1, lower+
    1)]
156     keep_rows = upper_rows + lower_rows
157     ab = ab[keep_rows, :]
158     return ab
159
160
161 def stopping_criteria(rms_new: float, rms_old: float, tol=
    1e-1) -> bool:
162     """
163     Determines whether or not the algorithm can stop.
    Currently evaluates against arbitrary conditions. To fully
164     integrate rtol and vtol into code, they need to be
    included in one of the params objects. Tests if the
    position and
165     velocity are within a certain distance of the previous
     iteration. Assuming with each step we get closer, this
166     implies we were within the specified tolerances
    supplied, or assumed above.
167
168     :param rms_new: New root mean square from current
    iteration
169     :param rms_old: Root mean square from previous
    iteration
170     :param tol: relative tolerance between updates
171     :return: returns if change is within relative
    tolerance
```

```
172     """
173     if rms_new == 0:
174         return True
175     percent_diff = np.abs((rms_old - rms_new)/rms_old)
176     return percent_diff < tol
177
```

```python
import enum


class Perturbations(enum.Enum):
    """
    List of acceptable perturbations to be included by the user for propagation purposes.
    """
    J2 = "J2"
    J3 = "J3"
    Drag = "Drag"
    SRP = "SRP"
    Moon = "Lunar Gravity"
    Sun = "Solar Gravity"


class Frames(enum.Enum):
    """
    List of acceptable frames of reference for physical locations.
    """
    ECI = "ECI"
    ECEF = "ECEF"
    LLA = "LLA"


class Angles(enum.Enum):
    """
    List of acceptable angle pairs.
    """
    Local = "Local"
    Celestial = "Celestial"

```

```python
from astropy.coordinates import GCRS, ITRS, ICRS, CIRS, CartesianRepresentation, EarthLocation
from astropy import units as u
from astropy.time import Time
import numpy as np
from typing import List


def lla_to_ecef(lla: List) -> np.ndarray:
    """
    Converts Lat, Lon, Alt to x,y,z position in ECEF
    :param lla: List of coordinate [lat, lon, alt]. Units [deg, deg, km]
    """
    loc = EarthLocation.from_geodetic(lat=lla[0], lon=lla[1], height=lla[2])
    r = np.array([loc.x.value, loc.y.value, loc.z.value])
    return r


def ecef_to_lla(r: np.ndarray) -> List:
    """
    Converts coordinate in ECEF frame to Lat and Lon
    :param r: Position in ECEF frame. Numpy array Units [km]
    """
    loc = EarthLocation.from_geocentric(r[0] * u.km, r[1] * u.km, r[2] * u.km)
    lat = loc.geodetic.lat
    lon = loc.geodetic.lon
    alt = loc.geodetic.height
    lla = [lat, lon, alt]
    return lla


def eci_to_ecef(r: np.ndarray, time: Time) -> np.ndarray:
    """
    Converts coordinates in Earth Centered Inertial frame to Earth Centered Earth Fixed.
    :param r: position of satellite in ECI frame. Units [km]
    :param time: Time of observation
    """
    gcrs = GCRS(CartesianRepresentation(r[0] * u.km, r[1] * u.km, r[2] * u.km), obstime=time)
    itrs = gcrs.transform_to(ITRS(obstime=time))
```

```
39      x_ecef = itrs.x.value
40      y_ecef = itrs.y.value
41      z_ecef = itrs.z.value
42      return np.array([x_ecef, y_ecef, z_ecef])
43
44
45  def ecef_to_eci(r: np.ndarray, time: Time) -> np.ndarray:
46      """
47      Converts coordinates in Earth Centered Earth Initial
    frame to Earth Centered Initial.
48      :param r: position of satellite in ECEF frame. Units [
    km]
49      :param time: Time of observation
50      """
51      itrs = ITRS(CartesianRepresentation(r[0] * u.km, r[1
    ] * u.km, r[2] * u.km), obstime=time)
52      gcrs = itrs.transform_to(GCRS(obstime=time))
53      x_eci = gcrs.cartesian.x.value
54      y_eci = gcrs.cartesian.y.value
55      z_eci = gcrs.cartesian.z.value
56      return np.array([x_eci, y_eci, z_eci])
57
58
59  def eci_to_icrs(r: np.ndarray, time: Time) -> np.ndarray:
60      gcrs = GCRS(CartesianRepresentation(r[0] * u.km, r[1
    ] * u.km, r[2] * u.km), obstime=time)
61      icrs = gcrs.transform_to(ICRS)
62      x_icrs = icrs.cartesian.x.value
63      y_icrs = icrs.cartesian.y.value
64      z_icrs = icrs.cartesian.z.value
65      return np.array([x_icrs, y_icrs, z_icrs])
66
67
68  def eci_to_angles(r: np.ndarray, time: Time) -> np.ndarray:
69      gcrs = GCRS(CartesianRepresentation(r[0] * u.km, r[1
    ] * u.km, r[2] * u.km), obstime=time)
70      ra = gcrs.ra.value
71      dec = gcrs.dec.value
72      return np.array([ra, dec])
73
74
75  def icrs_to_eci(r: np.ndarray, time: Time) -> np.ndarray:
76      icrs = ICRS(CartesianRepresentation(r[0] * u.km, r[1
    ] * u.km, r[2] * u.km))
77      gcrs = icrs.transform_to(GCRS)
78      x_gcrs = gcrs.cartesian.x.value
```

```
79      y_gcrs = gcrs.cartesian.y.value
80      z_gcrs = gcrs.cartesian.z.value
81      return np.array([x_gcrs, y_gcrs, z_gcrs])
82
```

```
1 import astropy.units as u
2
3 mu = 398600.44184 * u.km * u.km / u.s / u.s          # Units
   [km^2/s^2]
4 lunar_period = 27.3 * u.day                          # Units
   [day]
5 solar_period = 1 * u.year                            # Units
   [year]
6
```

```python
import numpy as np
from src.enums import Perturbations
from poliastro.twobody import Orbit
from poliastro.twobody.propagation import cowell
from poliastro.bodies import Earth
from poliastro.core.perturbations import J2_perturbation, atmospheric_drag_exponential, J3_perturbation, \
    radiation_pressure, third_body
from astropy import units as u
from astropy.time import Time
from src.dto import PropParams
from typing import Dict


def state_propagate(x: np.ndarray, epoch_obs: Time, params: PropParams) -> np.ndarray:
    """
    Propagates the state vector from moment of description to moment of observation in time another using the poliasto
    library. Allows for custom perturbations.
    :param x: State vector at time of original description
    :param epoch_obs: Time of observation.
    :param params: object which serves as catch all for relevant info. Includes dt or amount of time between initial
    description to moment of observation, epoch of initial description, and perturbations to be included.
    :return: Returns state vector at moment of observation
    """
    r = x[0:3] * u.km
    v = x[3:6] * u.km / u.s
    dt = epoch_obs - params.epoch

    sat_i = Orbit.from_vectors(Earth, r, v, epoch=params.epoch)
    sat_f = sat_i.propagate(dt, method=cowell, ad=a_d, perturbations=params.perturbations)
    output = np.concatenate([sat_f.r.value, sat_f.v.value])
    return output


def a_d(t0, state, k, perturbations: Dict):
    """
    Custom perturbation function that is passed directly to poliastro to be executed in their code, hence the need for
    summation() to be included within. Current structure
```

```
37 allows user to pick and chose which perturbations they
   would
38     like to include, requiring that the desired
   perturbation objects are created, filled, and passed.
39
40     Note: To improve upon existing perturbation functions
   or to add more, everything must be self-contained within
   the
41     function.
42
43     :param t0: Required by poliastro
44     :param state: Required by poliastro
45     :param k: Required by poliastro (gravitational
   parameter-mu)
46     :param perturbations: Dictionary of perturbations
   desired by the user. Keys correspond to the perturbations
   Enum
47     class in Enum.py, while values correspond to objects in
    the dto.py class.
48     :return: Returns a force that describes the impact of
   all desired perturbations
49     """
50     fun = []
51     if Perturbations.J2 in perturbations:
52         perturbation = perturbations.get(Perturbations.J2)
53         fun.append(J2_perturbation(t0, state, k,
   perturbation.J2, perturbation.R))
54     if Perturbations.Drag in perturbations:
55         perturbation = perturbations.get(Perturbations.Drag
   )
56         fun.append(atmospheric_drag_exponential(t0, state,
   k, perturbation.R, perturbation.C_D,
57
   perturbation.A/perturbation.m, perturbation.H0,
   perturbation.rho0))
58     if Perturbations.J3 in perturbations:
59         perturbation = perturbations.get(Perturbations.J3)
60         fun.append(J3_perturbation(t0, state, k,
   perturbation.J3, perturbation.R))
61     if Perturbations.SRP in perturbations:
62         perturbation = perturbations.get(Perturbations.SRP)
63         fun.append(radiation_pressure(t0, state, k,
   perturbation.R, perturbation.C_R, perturbation.A/
   perturbation.m,
64                                       perturbation.Wdivc_s
   , perturbation.star))
```

```
65      if Perturbations.Moon in perturbations:
66          perturbation = perturbations.get(Perturbations.
    Moon)
67          fun.append(third_body(t0, state, k, perturbation.
    k_third, perturbation.third_body))
68      if Perturbations.Sun in perturbations:
69          perturbation = perturbations.get(Perturbations.Sun
    )
70          fun.append(third_body(t0, state, k, perturbation.
    k_third, perturbation.third_body))
71
72      def summation(arr):
73          if len(arr) == 0:
74              return np.zeros(3)
75          output = arr[0]
76          for i in range(1, len(arr)):
77              output += arr[i]
78          return output
79
80      return summation(fun)
81
```

```python
from poliastro.ephem import build_ephem_interpolant
from astropy.coordinates import solar_system_ephemeris
from poliastro.bodies import Moon, Sun, Earth
from poliastro.constants import Wdivc_sun, H0_earth, rho0_earth
from src.dto import ThirdBody, J2, J3, SRP, Drag
from src.constants import solar_period, lunar_period
from astropy import units as u
from astropy.time import Time


solar_system_ephemeris.set("de432s")
R = Earth.R.to(u.km).value


def build_lunar_third_body(epoch: Time, rtol=1e-2) -> ThirdBody:
    """
    This function creates a callable moon object for third_body perturbation. Over long periods of integration, this
    may longer prove to be an accurate description. Needs to be investigated.
    :param epoch:   The time about which the interpolating function is created.
    :param rtol: determines number of points generated. Drives the time of execution significantly. Example online used
    rtol=1e-2. A smaller number is not accepted. Could be increased for more accuracy, that being said, the position of
    the Moon does not need to be that accurate.
    :return: Returns callable object that describes the Moon's position
    """
    epoch.format = "jd"
    k_moon = Moon.k.to(u.km ** 3 / u.s ** 2).value
    body_moon = build_ephem_interpolant(Moon, lunar_period, (epoch.value * u.day,

     epoch.value * u.day + 60 * u.day), rtol=rtol)
    return ThirdBody(k_moon, body_moon)


def build_solar_third_body(epoch: Time, rtol=1e-2) -> ThirdBody:
```

```
33      """
34      This function creates a callable Sun object for
    third_body and SRP perturbation. Over long periods of
    integration,
35      this may longer prove to be an accurate description.
    Needs to be investigated.
36      :param epoch:   The time about which the interpolating
    function is created.
37      :param rtol: determines number of points generated.
    Drives the time of execution significantly. Example online
    used
38      rtol=1e-2. A smaller number is not accepted. Could be
    increased for more accuracy, that being said, the position
    of
39      the Sun does not need to be that accurate.
40      :return: Returns callable object that describes the Sun
    's position
41      """
42      epoch.format = "jd"
43      k_sun = Sun.k.to(u.km ** 3 / u.s ** 2).value
44      body_sun = build_ephem_interpolant(Sun, solar_period, (
    epoch.value * u.day, epoch.value * u.day + 60 * u.day),
45                                         rtol=rtol)
46      return ThirdBody(k_sun, body_sun)
47
48
49  def build_j2() -> J2:
50      """
51      Builds J2 object used in propagation. Requires no input
     since all of the values are independent of orbital
    position
52      and time.
53      """
54      return J2(Earth.J2.value, R)
55
56
57  def build_j3() -> J3:
58      """
59      Build J3 object used in perturbation. Requires no input
     since all of the values are independent of orbital
    position
60      and time.
61      """
62      return J3(Earth.J3.value, R)
63
64
```

```
65 def build_srp(c_r, a, m, epoch, rtol=1e-2) -> SRP:
66     """
67     Build Solar Radiation Object used in perturbation.
68     :param c_r: Comparable to coefficient of Drag but for
   radiation pressure. Unitless
69     :param a: Cross sectional area exposed to radiation
   pressure. Units [m^2]
70     :param m: Mass of the satellite.  Units [kg]
71     :param epoch: Time required to interpolate solar
   position
72     :param rtol: determines number of points generated.
   Drives the time of execution significantly. Example online
    used
73     rtol=1e-2. A smaller number is not accepted. Could be
   increased for more accuracy, that being said, the position
    of
74     the Sun does not need to be that accurate.
75     """
76     epoch.format = "jd"
77     body_sun = build_ephem_interpolant(Sun, 1 * u.year, (
   epoch.value * u.day, epoch.value * u.day + 60 * u.day),
78                                        rtol=rtol)
79     return SRP(R, c_r, a, m, Wdivc_sun.value, body_sun)
80
81
82 def build_basic_drag(c_d, a, m):
83     """
84     Build Basic Atmospheric Drag Object used in
   perturbation.
85     :param c_d: Coefficient of Drag. Unitless
86     :param a: Cross-sectional area. Units [m^2]
87     :param m: Mass. Units [kg]
88     """
89     return Drag(R, c_d, a, m, H0_earth, rho0_earth)
90
```

```python
import numpy as np
import numpy.linalg as la
from src.dto import Observation
from src.enums import Frames


def y(x: np.ndarray, observation: Observation):
    """
    This function serves as a prediction function. It is used to describe the right ascension and declination of an observed satellite from an observer. Math is done in ECI Frame
    :param x: State Vector of satellite in ECI frame.
    :param observation: Parameters relevant to observation. Includes epoch, frame, and location of observation/observer.
    """
    r_obj = x[0:3]
    assert observation.frame == Frames.ECI
    rr = r_obj - observation.position
    alpha, dec = get_ra_and_dec(rr)
    return np.array([alpha, dec])


def get_ra_and_dec(rr: np.ndarray) -> np.ndarray:
    """
    Prediction function. Determines observational angles (WHICH) of the sateliite from observer.
    :param rr: Position of the spacecraft relative to observer
    """
    alpha = np.arctan2(rr[1], rr[0]) * 180 / np.pi
    dec = 90 - np.arccos(rr[2]/la.norm(rr)) * 180 / np.pi
    return np.array([alpha, dec])
```

```python
import numpy as np
from astropy.time import Time
from src.dto import PropParams
from src.state_propagator import state_propagate
from src.core import direction_isolator


def cov_propagate(x: np.ndarray, epoch_t: Time, prop_params: PropParams, p_i: np.ndarray) -> np.ndarray:
    """
    This function propagates a covariance matrix through time.

    :param x: state vector covariance matrix is tied to at original epoch
    :param epoch_t: Final epoch
    :param prop_params: Parameters relevant to propagation. Super accurate propagation is not always required. Some
    textbooks suggest using variational equations to save time.
    :param p_i: Covariance matrix at initial epoch
    """
    dr = .1
    dv = .005
    delta = np.array([dr, dr, dr, dv, dv, dv])

    a = dx_dx0(x, epoch_t, prop_params, delta)
    p_t = a @ p_i @ a.T
    return p_t


def dx_dx0(x: np.ndarray, epoch_t: Time, prop_params: PropParams, delta: np.ndarray) -> np.ndarray:
    """
    Calculates partial derivatives of a state vector in the future based on a current state vector.

    :param x: state vector at original epoch
    :param epoch_t: Final epoch
    :param prop_params: Parameters relevant to propagation
    :param delta: Array holding step sizes for derivatives
    """
    n = len(x)
    a = np.zeros((n, n))

    for j in range(0, n):
```

```
40          temp1 = state_propagate(x + direction_isolator(
   delta, j), epoch_t, prop_params)
41          temp2 = state_propagate(x - direction_isolator(
   delta, j), epoch_t, prop_params)
42          temp3 = temp1 - temp2
43
44          for i in range(0, n):
45              a[i][j] = temp3[i] / (2 * delta[i])
46      return a
47
```

```
''' 
Credit to the tle-tools library on github at https://github.com/FedericoStra/tletools. I was unable to install the package so I have copied the relevant code directly instead. There exist some small changes to the source material. Additional functionality has been added to meet my needs.

The module :mod:`tletools.tle` defines the :class:`TLE`.
whose attributes are quantities (:class:`astropy.units.Quantity`), a type able to represent
a value with an associated unit taken from :mod:`astropy.units`.
'''

import attr
import numpy as np
import astropy.units as u
from astropy.time import Time
from poliastro.twobody import Orbit
from poliastro.bodies import Earth
from src.interface.tle_util import parse_decimal, parse_float, m_to_nu, checksum, conv_year, DEG2RAD, RAD2DEG, rev, \
    convert_value_to_str, nu_to_m, ensure_positive_angle
from src.constants import mu
from typing import Tuple


@attr.s
class TLE:
    """Data class representing a single TLE.
    A two-line element set (TLE) is a data format encoding a list of orbital
    elements of an Earth-orbiting object for a given point in time, the epoch.
    All the attributes parsed from the TLE are expressed in the same units that
    are used in the TLE format.
    :ivar str name:
        Name of the satellite.
    :ivar str norad:
        NORAD catalog number (https://en.wikipedia.org/wiki/Satellite_Catalog_Number).
    :ivar str classification:
        'U', 'C', 'S' for unclassified, classified, secret.
    :ivar str int_desig:
```

```
37          International designator (https://en.wikipedia.org/
    wiki/International_Designator),
38      :ivar int epoch_year:
39          Year of the epoch.
40      :ivar float epoch_day:
41          Day of the year plus fraction of the day.
42      :ivar float dn_o2:
43          First time derivative of the mean motion divided by
     2.
44      :ivar float ddn_o6:
45          Second time derivative of the mean motion divided
    by 6.
46      :ivar float bstar:
47          BSTAR coefficient (https://en.wikipedia.org/wiki/
    BSTAR).
48      :ivar int set_num:
49          Element set number.
50      :ivar float inc:
51          Inclination.
52      :ivar float raan:
53          Right ascension of the ascending node.
54      :ivar float ecc:
55          Eccentricity.
56      :ivar float argp:
57          Argument of perigee.
58      :ivar float M:
59          Mean anomaly.
60      :ivar float n:
61          Mean motion.
62      :ivar int rev_num:
63          Revolution number.
64      (:class:`astropy.units.Quantity`), a type able to
    represent a value with
65      an associated unit taken from :mod:`astropy.units`.
66      """
67
68      # name of the satellite
69      name = attr.ib()
70      # NORAD catalog number (https://en.wikipedia.org/wiki/
    Satellite_Catalog_Number)
71      norad = attr.ib()
72      classification = attr.ib()
73      int_desig = attr.ib()
74      epoch_year = attr.ib(converter=conv_year)
75      epoch_day = attr.ib()
76      dn_o2 = attr.ib()
```

```
 77      ddn_o6 = attr.ib()
 78      bstar = attr.ib()
 79      set_num = attr.ib(converter=int)
 80      inc = attr.ib()
 81      raan = attr.ib()
 82      ecc = attr.ib()
 83      argp = attr.ib()
 84      m = attr.ib()
 85      n = attr.ib()
 86      rev_num = attr.ib(converter=int)
 87
 88      @property
 89      def a(self):
 90          """Semi-major axis."""
 91          return (mu.value / self.n.to_value(u.rad / u.s
    ) ** 2) ** (1 / 3) * u.km
 92
 93      @property
 94      def epoch(self):
 95          return Time(self.epoch_year + self.epoch_day/365.
    25, format='decimalyear', scale='utc')
 96
 97      @property
 98      def nu(self):
 99          """True anomaly."""
100          thing = self.m
101          m = ((self.m.value + 180) % 360 - 180) * DEG2RAD
102          return m_to_nu(m, self.ecc) * RAD2DEG * u.deg
103
104      @property
105      def period(self):
106          """Period"""
107          return 1/self.n.to(rev / u.s) * rev
108
109      @classmethod
110      def from_lines(cls, tle_string):
111          """
112          Creates a tle object form a TLE string, requires
    name as the first line. 3 lines total
113
114          :param tle_string: TLE string from database
115          """
116          tle_lines = tle_string.strip().splitlines()
117          name = tle_lines[0]
118          line1 = tle_lines[1]
119          line2 = tle_lines[2]
```

```
120
121         return cls(
122             name=name,
123             norad=line1[2:7],
124             classification=line1[7],
125             int_desig=line1[9:17],
126             epoch_year=line1[18:20],
127             epoch_day=float(line1[20:32]),
128             dn_o2=str(line1[33:43]),    # altered to be a
    str
129             ddn_o6=str(line1[44:52]),   # altered to be a
    str
130             bstar=str(line1[53:61]),    # altered to be a
    str
131             set_num=line1[64:68],
132             inc=u.Quantity(float(line2[8:16]), u.deg),
133             raan=u.Quantity(float(line2[17:25]), u.deg),
134             ecc=u.Quantity(parse_decimal(line2[26:33]), u.
    one),
135             argp=u.Quantity(float(line2[34:42]), u.deg),
136             m=u.Quantity(float(line2[43:51]), u.deg),
137             n=u.Quantity(float(line2[51:63]), rev / u.day
    ),
138             rev_num=line2[63:68])
139
140     def to_orbit(self, attractor=Earth) -> Orbit:
141         """Convert to an orbit around the Earth."""
142         return Orbit.from_classical(
143             attractor=attractor,
144             a=self.a,
145             ecc=self.ecc,
146             inc=self.inc,
147             raan=self.raan,
148             argp=self.argp,
149             nu=self.nu,
150             epoch=self.epoch)
151
152     def update(self, x: np.ndarray, new_epoch: Time):
153         """ Updates values of the TLE according to a new
    estimation"""
154
155         new_epoch.format = "decimalyear"
156         r = x[0:3] * u.km
157         v = x[3:6] * u.km / u.s
158         obj = Orbit.from_vectors(Earth, r, v, epoch=
    new_epoch)
```

```
159
160         dt = (new_epoch - self.epoch).to(u.s)
161         delta_m = dt * self.n.to(u.deg / u.s)
162         new_revs = int((self.m + delta_m) / (360 * u.deg))
163         self.rev_num += new_revs
164
165         self.epoch_year = int(new_epoch.decimalyear)
166         self.epoch_day = (new_epoch.decimalyear % 1) * 365
    .25
167         self.inc = ensure_positive_angle(obj.inc)
168         self.raan = ensure_positive_angle(obj.raan)
169         self.ecc = obj.ecc
170         self.argp = ensure_positive_angle(obj.argp)
171         self.m = ensure_positive_angle(nu_to_m(obj.nu, obj
    .ecc).to(u.deg))
172         self.n = obj.n.to(rev / u.d)
173         self.set_num += 1
174         return self
175
176     def to_string(self):
177         """Convert TLE into a string"""
178
179         self.epoch.format = "decimalyear"
180         epoch_year = str(int(self.epoch.decimalyear))[2:4]
181         epoch_day = str(convert_value_to_str((self.epoch.
    value % 1) * 365.25, 3, 8))
182
183         set_num = convert_value_to_str(self.set_num, 4, 0
    )[0:4]
184         inc = convert_value_to_str(self.inc.to(u.deg).
    value, 3, 4)
185         raan = convert_value_to_str(self.raan.to(u.deg).
    value, 3, 4)
186         ecc = convert_value_to_str(self.ecc.value, 1, 7)[2
    :]
187         argp = convert_value_to_str(self.argp.to(u.deg).
    value, 3, 4)
188         m = convert_value_to_str(self.m.to(u.deg).value, 3
    , 4)
189         n = convert_value_to_str(self.n.value, 2, 8)
190         rev_num = convert_value_to_str(self.rev_num, 5, 0
    )[0:5]
191
192         line0 = self.name
193         line1 = "\n1 " + self.norad + self.classification
     + " " + self.int_desig + " " + epoch_year + epoch_day +
```

```
193 " " \
194                 + self.dn_o2 + " " + self.ddn_o6 + " " +
    self.bstar + " 0 " + set_num
195         line2 = "\n2 " + self.norad + " " + inc + " " +
    raan + " " + ecc + " " + argp + " " + m + " " + n +
    rev_num
196         return line0 + line1 + checksum(line1[1:]) + line2
     + checksum(line2[1:])
197
198     def to_state(self) -> Tuple[np.ndarray, Time]:
199         """Converts a TLE into a functional state (ECI)
    and epoch"""
200         obj = self.to_orbit()
201         x = np.concatenate([obj.r.value, obj.v.value])
202         return x, self.epoch
203
```

```python
from src.dto import Observation
from src.enums import Frames
from src.frames import lla_to_ecef, ecef_to_eci
import astropy.units as u


def convert_obs_from_lla_to_ecef(observation: Observation) -> Observation:
    """
    Converts Observer location from LLA to ECEF frame before calculations to limit total computational cost.

    :param observation: Observational params relevant to prediction function
    """
    assert observation.frame == Frames.LLA
    observation.frame = Frames.ECEF
    observation.position = lla_to_ecef(observation.position)
    return observation


def convert_obs_from_ecef_to_eci(observation: Observation) -> Observation:
    """
    Converts Observer location from ECEF to ECI frame before calculations to limit total computational cost.

    :param observation: Observational params relevant to prediction function
    """
    assert observation.frame == Frames.ECEF
    observation.frame = Frames.ECI
    observation.position = ecef_to_eci(observation.position, observation.epoch)
    return observation


def convert_obs_from_lla_to_eci(obs_params: Observation) -> Observation:
    """
    Converts Observer location from LLA to ECI frame before calculations to limit total computational cost.

    :param obs_params: Observational params relevant to prediction function
```

```
36      """
37      assert obs_params.frame == Frames.LLA
38      obs_params.frame = Frames.ECI
39      obs_params.position = ecef_to_eci(lla_to_ecef(
    obs_params.position), obs_params.epoch)
40      return obs_params
41
42
43  def verify_locational_units(obs_params: Observation) ->
    Observation:
44      """
45      Units are assumed to be a certain set later down the
    pipeline. THis function serves to ensure that the correct
    units
46      as being assumed.
47      :param obs_params: Observation units relevant to the
    prediction function
48      """
49      lla_units = [u.deg, u.deg, u.km]
50      spacial_units = [u.km, u.km, u.km]
51      if obs_params.frame == Frames.LLA:
52          desired_units = lla_units
53      else:
54          assert(obs_params.frame == Frames.ECI or Frames.
    ECEF)
55          desired_units = spacial_units
56      for i in range(3):
57          if obs_params.position[i].unit is not desired_units
    [i]:
58              obs_params.position[i] = obs_params.position[i
    ].to(desired_units[i])
59      return obs_params
60
```

```python
import string
import numpy as np
import astropy.units as u
from poliastro.core.angles import M_to_E, E_to_nu, nu_to_E
, E_to_M

DEG2RAD = np.pi / 180.
RAD2DEG = 180. / np.pi

rev = u.def_unit(
    ['rev', 'revolution'],
    2.0 * np.pi * u.rad,
    prefixes=False,
    doc="revolution: angular measurement, a full turn or
rotation")
u.add_enabled_units(rev)


def conv_year(s):
    """Interpret a two-digit year string."""
    if isinstance(s, int):
        return s
    y = int(s)
    return y + (1900 if y >= 57 else 2000)


def parse_decimal(s):
    """Parse a floating point with implicit leading dot.
    """
    return float('.' + s)


def parse_float(s):
    """Parse a floating point with implicit dot and
exponential notation.
    """
    return float(s[0] + '.' + s[1:6] + 'e' + s[6:8])


def m_to_nu(m, ecc):
    """True anomaly from mean anomaly.
    :param float m: Mean anomaly in radians.
    :param float ecc: Eccentricity.
    :returns: `nu`, the true anomaly, between -π and π
radians.

```

```
43      **Warning**
44      The mean anomaly must be between -π and π radians.
45      The eccentricity must be less than 1.
46      """
47      return E_to_nu(M_to_E(m, ecc), ecc)
48
49
50  def nu_to_m(nu, ecc):
51      """Mean anomaly from true anomaly.
52      :param float nu: True anomaly in radians.
53      :param float ecc: Eccentricity.
54      :returns: `nu`, the true anomaly, between -π and π
    radians.
55
56      **Warning**
57      The mean anomaly must be between -π and π radians.
58      The eccentricity must be less than 1.
59      """
60      return E_to_M(nu_to_E(nu.to(u.rad), ecc), ecc) * u.rad
61
62
63  def checksum(line: str) -> str:
64      """
65      Check sum function for TLEs. Adds all non-letters as
    their value and - signs as 1.
66
67      :param line: A line of a TLE
68      """
69      L = line.strip()
70      cksum = 0
71      for i in range(68):
72          c = L[i]
73          if c == " " or c == "." or c == "+" or c in string.
    ascii_letters:
74              continue
75          elif c == "-":
76              cksum = cksum + 1
77          else:
78              cksum = cksum + int(c)
79      cksum %= 10
80      return str(cksum)
81
82
83  def convert_value_to_str(value, length_before, length_after
    ) -> str:
84      """
```

```
 85      Converts a value into a string with appropriate
    formatting. Allows user to decide how many characters are
    before and
 86      after the decimal point.
 87
 88      :param value: the value to be formatted
 89      :param length_before: Number of characters before the
    decimal
 90      :param length_after: Number of characters after the
    decimal
 91      """
 92      thing = value % 1
 93      after = round(value % 1, length_after)
 94      before = round(value - after)
 95      before_string = str(int(before))
 96      after_string = str(after)[2:]
 97      while len(before_string) < length_before:
 98          before_string = " " + before_string
 99      while len(after_string) < length_after:
100          after_string += "0"
101      output = before_string[0:length_before] + "." +
    after_string[0:length_after]
102      return output
103
104
105  def ensure_positive_angle(angle):
106      angle = angle.to(u.deg)
107      while angle.value < 0:
108          angle += (360 * u.deg)
109      while angle.value > 360:
110          angle -= (360 * u.deg)
111      return angle
112
```

```python
import numpy as np
from src.state_propagator import state_propagate
from src.dto import PropParams
from src.enums import Frames
from src.frames import lla_to_ecef, eci_to_ecef, ecef_to_eci
from typing import List
from astropy.time import Time
from astropy.coordinates import Angle


def local_angles(rr: np.ndarray, lla: List) -> np.ndarray:
    """
    Gives local azimuth and zenith angles for a satellite with respect to an observer's position on the Earth. Azimuth
    is measured from local North, where positive indicates Eastward or clockwise.

    :param rr: observational difference vector in ECEF frame. Units [km]
    :param lla: List of [lat, long, altitude]. Units [deg, deg, km]
    """
    rot_mat = rotation_matrix(lla[0].value, lla[1].value)
    local_sky = rot_mat.T @ rr

    el = np.arcsin(local_sky[2] / np.linalg.norm(local_sky)) * 180 / np.pi
    az = 90 - np.arctan2(-local_sky[0], local_sky[1]) * 180 / np.pi

    return np.array([az, el])


def ry(lat: float) -> np.ndarray:
    """
    Gives rotation matrix to rotate around the y-axis in ECEF frame to align the z-axis with the observers latitude

    :param lat: Latitude of the observer. Units [deg]
    """
    angle = -lat * np.pi / 180
    c = np.cos(angle)
    s = np.sin(angle)
    a = np.array([[-s, 0, c], [0, 1, 0], [-c, 0, -s]])
```

```
38      return a
39
40
41  def rz(lon: float) -> np.ndarray:
42      """
43      Gives rotation matrix to rotate around the z-axis in
    ECEF from to align x-axis with observers longitude
44
45      :param lon: Longitude of the observer. Units [deg]
46      """
47      angle = lon * np.pi / 180
48      c = np.cos(angle)
49      s = np.sin(angle)
50      a = np.array([[c, -s, 0], [s, c, 0], [0, 0, 1]])
51      return a
52
53
54  def rotation_matrix(lat, lon):
55      """
56      Build rotation matrix from ECEF z-axis along direction
    of observer. This transpose of this matrix converts an
57      observational difference vector into an x, y, z
    coordinate system x and y are perpendicular components of
    the
58      observation vector and z is the parallel component. Z
    gives zenith angle while x,y give azimuth angle. Notably,
59      in this frame x points south and y points east.
60
61      :param lat: Latitude of the observer
62      :param lon: Longitude of the observer
63      """
64      return rz(lon) @ ry(lat)
65
66  #
67  # def get_local_angles_via_state_propagation(x: np.ndarray
    , prop_params: PropParams, epoch_i: Time, epoch_f: Time, n
    : int,
68  #                                            obs_pos_lla,
    obs_frame: Frames):
69  #     """
70  #     This function returns a list of [theta, phi, Time]
    from initial epoch in prop_params to final epoch, with an
71  #     n points between those two epochs.
72  #
73  #     :param x: State vector at initial epoch
74  #     :param prop_params: Parameters relevant to
```

```
74 propagation. Includes initial
75 #     :param epoch_i: Epoch of first desired time
76 #     :param epoch_f: Epoch of final desired time
77 #     :param n: Number of desired points between initial
   and final epoch (Does not include those two)
78 #     :param obs_pos_lla: Observer location. Accepts list
   for LLA. [lat * u.deg, lon * u.deg, alt * u.km]
79 #     :param obs_frame: Frame observer location is in.
   Accepts Frames.LLA
80 #     """
81 #     assert obs_frame == Frames.LLA
82 #     obs_pos_ecef = lla_to_ecef(obs_pos_lla)
83 #
84 #     dt = (epoch_f - epoch_i) / (n+1)
85 #     output = []
86 #     for i in range(0, n + 2):
87 #         desired_epoch = epoch_i + dt * i
88 #         obj_pos_eci = state_propagate(x, desired_epoch,
   prop_params)[0:3]
89 #         obj_pos_ecef = eci_to_ecef(obj_pos_eci,
   desired_epoch)
90 #         rr = obj_pos_ecef - obs_pos_ecef
91 #         angles = local_angles(rr, obs_pos_lla)
92 #         output.append([angles[0], angles[1],
   desired_epoch])
93 #     return output
94
```

```python
import re


def dms_to_dd(input_string: str) -> float:
    """
    Converts string value of degree minute seconds into a decimal degree float.

    :param input_string: string following the form "degree minute' seconds"
    :return: decimal degree float
    """
    input_string += " "
    try:
        degrees = re.search(r"[\d\.]+ ", input_string).group(0).replace(" ", "")
    except AttributeError:
        degrees = 0
    try:
        minutes = re.search(r"[\d\.]+\'", input_string).group(0).replace("\'", "")
    except AttributeError:
        minutes = 0
    try:
        seconds = re.search(r"[\d\.]+\"", input_string).group(0).replace("\"", "")
    except AttributeError:
        seconds = 0
    return float(degrees) + float(minutes) / 60 + float(seconds) / 3600

```

```
<MainWindow>:
    name: "main"
    state_label: state_label
    epoch_label: epoch_label
    params_label: params_label
    delta_x_apr_label: delta_x_apr_label
    p_apr_label: p_apr_label
    p_apr_value: p_apr_value
    observations_output: observations_output
    tag: tag

    FloatLayout:
        Label:
            id: state_label
            text_size: self.size
            halign: 'left'
            pos_hint:{"x": .1, "top": 1.1}
            size_hint:0.6, 0.2

        Label:
            text_size: self.size
            halign: 'left'
            id: epoch_label
            pos_hint:{"x": 0.1, "top":1}
            size_hint:0.6, 0.2

        Label:
            text_size: self.size
            halign: 'left'
            id: params_label
            pos_hint:{"x": .1, "top":0.90}
            size_hint:0.6, 0.2

        Button:
            pos_hint:{"x":0.75, "y": .8}
            size_hint:0.2,0.1
            text: "Edit core values"
            on_release:
                app.root.current = "addcore"
                root.manager.transition.direction = "left"

        Label:
            text_size: self.size
            halign: 'left'
            id: delta_x_apr_label
            pos_hint:{"x": 0.1, "y": .6}
```

```
47              size_hint:0.6, 0.2
48
49          Label:
50              text_size: self.size
51              halign: 'left'
52              id: p_apr_label
53              pos_hint:{"x": .1, "y":0.45}
54              size_hint:0.8, 0.2
55
56          Label:
57              text_size: self.size
58              halign: 'left'
59              valing: 'middle'
60              id: p_apr_value
61              pos_hint:{"x": .3, "y":0.35}
62              size_hint:0.8, 0.4
63
64          Button:
65              pos_hint:{"x":0.75, "y": .65}
66              size_hint:0.2,0.1
67              text: "Edit a priori values"
68              on_release:
69                  app.root.current = "apr"
70                  root.manager.transition.direction = "left"
71
72          Label:
73              text_size: self.size
74              halign: 'left'
75              pos_hint:{"x": .1, "y":0.25}
76              size_hint:0.2, 0.2
77              text: "Observations:    "
78
79          ScrollView:
80              do_scroll_x: True
81              do_scroll_y: True
82              pos_hint:{"x": .22, "y":0.1}
83              size_hint:0.45, 0.2
84              border_color: 'red'
85              Label:
86                  halign: 'left'
87                  id: observations_output
88                  size_hint_y: None
89                  height: self.texture_size[1]
90
91          Button:
92              pos_hint:{"x":0.75, "y": .5}
```

```
 93             size_hint:0.2,0.1
 94             text: "Edit observations"
 95             on_release:
 96                 app.root.current = "obs"
 97                 root.manager.transition.direction = "left"
 98 
 99         Button:
100             pos_hint:{"x":0.75, "y": 0.35}
101             size_hint:0.2,0.1
102             text: "Clear all values"
103             on_release:
104                 root.set_default_values()
105 
106         Button:
107             pos_hint:{"x":0.75, "y": 0.2}
108             size_hint:0.2,0.1
109             text: "Run"
110             on_release:
111                 root.run_clicked()
112                 root.manager.transition.direction = "up"
113 
114         Label:
115             id: tag
116             text_size: self.size
117             halign: 'right'
118             pos_hint:{"x": .6, "y": .0}
119             size_hint:0.4, 0.05
120             text: "Developed by Austin Ogle  Ver. 0.0.1  7
    /29/2020"
121 
122 
123 <AddCoreValues>:
124     name: "core_values"
125     state: state
126     epoch: epoch
127     tle: tle
128     is_isot: is_isot
129     is_jd: is_jd
130     tle_active: tle_active
131     include_j2: include_j2
132     include_j3: include_j3
133     include_solar: include_solar
134     include_lunar: include_lunar
135     include_srp: include_srp
136     include_drag: include_drag
137 
```

```
138     drag_A: drag_A
139     drag_C_d: drag_C_d
140     drag_m: drag_m
141     srp_A: srp_A
142     srp_C_r: srp_C_r
143     srp_m: srp_m
144 
145 
146     FloatLayout:
147         Label:
148             text_size: self.size
149             halign: 'left'
150             valign: 'middle'
151             pos_hint:{"x": 0.1, "top":0.95}
152             size_hint: 0.1, 0.1
153             text: "state: "
154 
155         TextInput:
156             id: state
157             font_size: (root.width**2 + root.height**2) /
    13**4 / 2
158             multiline: False
159             pos_hint: {"x": 0.2, "top":0.95}
160             size_hint: 0.6, 0.1
161             hint_text: 'r_x, r_y, r_z, v_x, v_y, v_z'
162 
163         Label:
164             text_size: self.size
165             halign: 'left'
166             valign: 'middle'
167             pos_hint:{"x": 0.1, "top":0.8}
168             size_hint: 0.1, 0.1
169             text: "epoch: "
170 
171         TextInput:
172             id: epoch
173             font_size: (root.width**2 + root.height**2) /
    13**4 / 2
174             multiline: False
175             pos_hint: {"x": 0.2, "top":0.8}
176             size_hint: 0.4, 0.1
177             hint_text: 'YYYY-MM-DDTHH:MM:SS.SSS'
178 
179         CheckBox:
180             id: is_isot
181             pos_hint: {"x": 0.65, "top":0.8}
```

```
182             size_hint: 0.05, 0.05
183             group: "opts"
184             on_active:
185                 root.isot_active()
186 
187         Label:
188             text_size: self.size
189             halign: 'left'
190             valign: 'middle'
191             pos_hint:{"x": 0.7, "top":0.8}
192             size_hint: 0.1, 0.05
193             text: "ISOT"
194 
195         CheckBox:
196             id: is_jd
197             pos_hint: {"x": 0.65, "top":0.75}
198             size_hint: 0.05, 0.05
199             group: "opts"
200             on_active:
201                 root.jd_active()
202 
203         Label:
204             text_size: self.size
205             halign: 'left'
206             valign: 'middle'
207             pos_hint:{"x": 0.7, "top":0.75}
208             size_hint: 0.1, 0.05
209             text: "JD"
210 
211 
212         Label:
213             text_size: self.size
214             halign: 'left'
215             valign: 'middle'
216             pos_hint:{"x": 0.05, "top":0.65}
217             size_hint: 0.15, 0.1
218             text: "Included Perturbations: "
219 
220         CheckBox:
221             id: include_j2
222             pos_hint: {"x": 0.2, "top":0.65}
223             size_hint: 0.05, 0.05
224             on_active:
225                 root.j2_active(self.active)
226 
227         Label:
```

```
228             text_size: self.size
229             halign: 'left'
230             valign: 'middle'
231             pos_hint:{"x": 0.25, "top":0.65}
232             size_hint: 0.1, 0.05
233             text: "J2"
234 
235         CheckBox:
236             id: include_j3
237             pos_hint: {"x": 0.2, "top":0.6}
238             size_hint: 0.05, 0.05
239             on_active:
240                 root.j3_active(self.active)
241 
242         Label:
243             text_size: self.size
244             halign: 'left'
245             valign: 'middle'
246             pos_hint:{"x": 0.25, "top":0.6}
247             size_hint: 0.1, 0.05
248             text: "J3"
249 
250         CheckBox:
251             id: include_solar
252             pos_hint: {"x": 0.30, "top":0.65}
253             size_hint: 0.05, 0.05
254             on_active:
255                 root.solar_active(self.active)
256 
257         Label:
258             text_size: self.size
259             halign: 'left'
260             valign: 'middle'
261             pos_hint:{"x": 0.35, "top":0.65}
262             size_hint: 0.2, 0.05
263             text: "Solar Gravity"
264 
265         CheckBox:
266             id: include_lunar
267             pos_hint: {"x": 0.30, "top":0.6}
268             size_hint: 0.05, 0.05
269             on_active:
270                 root.lunar_active(self.active)
271 
272         Label:
273             text_size: self.size
```

```
274             halign: 'left'
275             valign: 'middle'
276             pos_hint:{"x": 0.35, "top":0.6}
277             size_hint: 0.2, 0.05
278             text: "Lunar Gravity"
279 
280         CheckBox:
281             id: include_drag
282             pos_hint: {"x": 0.50, "top":0.65}
283             size_hint: 0.05, 0.05
284             on_active:
285                 root.drag_active(self.active)
286 
287         Label:
288             text_size: self.size
289             halign: 'left'
290             valign: 'middle'
291             pos_hint:{"x": 0.55, "top":0.65}
292             size_hint: 0.1, 0.05
293             text: "Drag"
294 
295         TextInput:
296             id: drag_A
297             font_size: (root.width**2 + root.height**2) /
    13**4 / 2.6
298             multiline: False
299             pos_hint: {"x": 0.6, "top":0.65}
300             size_hint: 0.1, 0.05
301             readonly: True
302             background_color: .6, .6, .6, 1
303             hint_text: "Area [m^2]"
304             hint_text_color: [1, 1, 1, 1]
305 
306         TextInput:
307             id: drag_C_d
308             font_size: (root.width**2 + root.height**2) /
    13**4 / 2.6
309             multiline: False
310             pos_hint: {"x": 0.72, "top":0.65}
311             size_hint: 0.1, 0.05
312             readonly: True
313             background_color: .6, .6, .6, 1
314             hint_text: "C_d"
315             hint_text_color: [1, 1, 1, 1]
316 
317         TextInput:
```

```
318             id: drag_C_d
319             font_size: (root.width**2 + root.height**2) /
    13**4 / 2.6
320             multiline: False
321             pos_hint: {"x": 0.72, "top":0.65}
322             size_hint: 0.1, 0.05
323             readonly: True
324             background_color: .6, .6, .6, 1
325             hint_text: "C_d"
326             hint_text_color: [1, 1, 1, 1]
327
328         TextInput:
329             id: drag_m
330             font_size: (root.width**2 + root.height**2) /
    13**4 / 2.6
331             multiline: False
332             pos_hint: {"x": 0.84, "top":0.65}
333             size_hint: 0.1, 0.05
334             readonly: True
335             background_color: .6, .6, .6, 1
336             hint_text: "Mass [kg]"
337             hint_text_color: [1, 1, 1, 1]
338
339
340         CheckBox:
341             id: include_srp
342             pos_hint: {"x": 0.50, "top":0.6}
343             size_hint: 0.05, 0.05
344             on_active:
345                 root.srp_active(self.active)
346
347         Label:
348             text_size: self.size
349             halign: 'left'
350             valign: 'middle'
351             pos_hint:{"x": 0.55, "top":0.6}
352             size_hint: 0.2, 0.05
353             text: "SRP"
354
355         TextInput:
356             id: srp_A
357             font_size: (root.width**2 + root.height**2) /
    13**4 / 2.6
358             multiline: False
359             pos_hint: {"x": 0.6, "top":0.60}
360             size_hint: 0.1, 0.05
```

```
361             readonly: True
362             background_color: .6, .6, .6, 1
363             hint_text: "Area [m^2]"
364             hint_text_color: [1, 1, 1, 1]
365 
366         TextInput:
367             id: srp_C_r
368             font_size: (root.width**2 + root.height**2) /
    13**4 / 2.6
369             multiline: False
370             pos_hint: {"x": 0.72, "top":0.60}
371             size_hint: 0.1, 0.05
372             readonly: True
373             background_color: .6, .6, .6, 1
374             hint_text: "C_d"
375             hint_text_color: [1, 1, 1, 1]
376 
377         TextInput:
378             id: srp_C_r
379             font_size: (root.width**2 + root.height**2) /
    13**4 / 2.6
380             multiline: False
381             pos_hint: {"x": 0.72, "top":0.6}
382             size_hint: 0.1, 0.05
383             readonly: True
384             background_color: .6, .6, .6, 1
385             hint_text: "C_r"
386             hint_text_color: [1, 1, 1, 1]
387 
388         TextInput:
389             id: srp_m
390             font_size: (root.width**2 + root.height**2) /
    13**4 / 2.6
391             multiline: False
392             pos_hint: {"x": 0.84, "top":0.60}
393             size_hint: 0.1, 0.05
394             readonly: True
395             background_color: .6, .6, .6, 1
396             hint_text: "Mass [km]"
397             hint_text_color: [1, 1, 1, 1]
398 
399         CheckBox:
400             id: tle_active
401             pos_hint:{"x": 0.2, "top":0.52}
402             size_hint: 0.05, 0.1
403             on_active:
```

```
404                 root.tle_state(self.active)
405
406         Label:
407             text_size: self.size
408             halign: 'left'
409             valign: 'middle'
410             pos_hint:{"x": 0.25, "top":0.52}
411             size_hint: 0.65, 0.1
412             text: "Use TLE input over state and epoch"
413
414         Label:
415             text_size: self.size
416             halign: 'left'
417             valign: 'middle'
418             pos_hint:{"x": 0.05, "top":0.4}
419             size_hint: 0.15, 0.1
420             text: "TLE: "
421
422         TextInput:
423             id: tle
424             font_size: (root.width**2 + root.height**2) /
    13**4 / 2.6
425             multiline: True
426             pos_hint: {"x": 0.2, "top":0.4}
427             size_hint: 0.6, 0.15
428             readonly: True
429             background_color: .6, .6, .6, 1
430             hint_text: "This box is read only until the
    checkbox has been activated"
431             hint_text_color: [1, 1, 1, 1]
432
433         Button:
434             pos_hint:{"x":0.2, "y": 0.1}
435             size_hint:0.6,0.1
436             text: "Update and return to main page"
437             on_release:
438                 root.update_values()
439                 root.manager.transition.direction = "right
    "
440
441 <AddAPrioriValues>:
442     name: "obs"
443     state: state
444     p_apr: p_apr
445
446     FloatLayout:
```

```
447         Label:
448             pos_hint:{"x": 0.1, "top":0.95}
449             size_hint: 0.2, 0.1
450             text: "\u0394 x (a priori): "
451 
452         TextInput:
453             id: state
454             font_size: (root.width**2 + root.height**2) /
    13**4  / 2
455             multiline: False
456             pos_hint: {"x": 0.3, "y":0.85}
457             size_hint: 0.6, 0.1
458             hint_text: 'r_x, r_y, r_z, v_x, v_y, v_z'
459 
460         Label:
461             pos_hint:{"x": 0.1, "top":0.55}
462             size_hint: 0.2, 0.1
463             text: "Covariance (a priori): "
464 
465         TextInput:
466             id: p_apr
467             font_size: (root.width**2 + root.height**2) /
    13**4 / 2
468             multiline: True
469             pos_hint: {"x": 0.4, "y":0.3}
470             size_hint: 0.4, 0.4
471             hint_text: "P11, P12, P13, P14, P15, P16\nP21
    , P22, P23, P24, P25, P26\nP31, P32, P33, P34, P35, P36\
    nP41, P42, P43, P44, P45, P46\nP51, P52, P53, P54, P55,
    P56\nP61, P62, P63, P64, P65, P66"
472 
473         Button:
474             pos_hint:{"x":0.55, "y": 0.1}
475             size_hint:0.3,0.1
476             text: "Update and return to main page"
477             on_release:
478                 root.update_values()
479                 root.manager.transition.direction = "right
    "
480 
481         Button:
482             pos_hint:{"x":0.15, "y": 0.1}
483             size_hint:0.3,0.1
484             text: "Clear values"
485             color: 1, 0, 0, 1
486             on_release:
```

```
487                 root.clear_values()
488 
489 <AddObservation>:
490     name: "obs"
491     lat: lat
492     lon: lon
493     alt: alt
494     epoch: epoch
495     ra: ra
496     dec: dec
497     sigma_ra: sigma_ra
498     sigma_dec: sigma_dec
499     observation_output: observation_output
500     is_isot: is_isot
501     is_jd: is_jd
502     is_dd: is_dd
503     is_dms: is_dms
504 
505     FloatLayout:
506         Label:
507             text_size: self.size
508             halign: 'left'
509             valign: 'middle'
510             pos_hint:{"x": 0.08, "y":0.85}
511             size_hint: 0.1, 0.1
512             text: "Observer \n lat, lon, alt: "
513 
514         TextInput:
515             id: lat
516             font_size: (root.width**2 + root.height**2) /
    13**4 / 2
517             multiline: False
518             pos_hint: {"x": 0.2, "y":0.85}
519             size_hint: 0.12, 0.1
520             hint_text: "00.000"
521 
522         TextInput:
523             id: lon
524             font_size: (root.width**2 + root.height**2) /
    13**4 / 2
525             multiline: False
526             pos_hint: {"x": 0.34, "y":0.85}
527             size_hint: 0.12, 0.1
528             hint_text: "00.000"
529 
530         CheckBox:
```

```
531             id: is_dd
532             pos_hint: {"x": 0.47, "y":0.90}
533             size_hint: 0.05, 0.05
534             group: "latlon"
535             on_active:
536                 root.dd_active()
537 
538         Label:
539             text_size: self.size
540             halign: 'left'
541             valign: 'middle'
542             pos_hint:{"x": 0.52, "y":0.90}
543             size_hint: 0.1, 0.05
544             text: "DD"
545 
546         CheckBox:
547             id: is_dms
548             pos_hint: {"x": 0.47, "y":0.85}
549             size_hint: 0.05, 0.05
550             group: "latlon"
551             on_active:
552                 root.dms_active()
553 
554         Label:
555             text_size: self.size
556             halign: 'left'
557             valign: 'middle'
558             pos_hint:{"x": 0.52, "y":0.85}
559             size_hint: 0.1, 0.05
560             text: "DMS"
561 
562         TextInput:
563             id: alt
564             font_size: (root.width**2 + root.height**2) /
    13**4 / 2
565             multiline: False
566             pos_hint: {"x": 0.6, "y":0.85}
567             size_hint: 0.1, 0.1
568             hint_text: "[km]"
569 
570         Label:
571             text_size: self.size
572             halign: 'left'
573             valign: 'middle'
574             pos_hint:{"x": 0.1, "y":0.7}
575             size_hint: 0.1, 0.1
```

```
576             text: "Epoch: "
577 
578         TextInput:
579             id: epoch
580             font_size: (root.width**2 + root.height**2) /
    13**4 / 2
581             multiline: False
582             pos_hint: {"x": 0.2, "y":0.7}
583             size_hint: 0.4, 0.1
584             hint_text: "YYYY-MM-DDTHH:MM:SS.SSS"
585 
586         CheckBox:
587             id: is_isot
588             pos_hint: {"x": 0.62, "y":0.75}
589             size_hint: 0.05, 0.05
590             group: "opts"
591             on_active:
592                 root.isot_active()
593 
594         Label:
595             text_size: self.size
596             halign: 'left'
597             valign: 'middle'
598             pos_hint:{"x": 0.67, "y":0.75}
599             size_hint: 0.1, 0.05
600             text: "ISOT"
601 
602         CheckBox:
603             id: is_jd
604             pos_hint: {"x": 0.62, "y":0.7}
605             size_hint: 0.05, 0.05
606             group: "opts"
607             on_active:
608                 root.jd_active()
609 
610         Label:
611             text_size: self.size
612             halign: 'left'
613             valign: 'middle'
614             pos_hint:{"x": 0.67, "y":0.7}
615             size_hint: 0.1, 0.05
616             text: "JD"
617 
618         Label:
619             pos_hint:{"x": 0.1, "y":0.55}
620             size_hint: 0.05, 0.1
```

```
621             text: "Measurements \n(\u03B1, \u03B4): "
622 
623         TextInput:
624             id: ra
625             font_size: (root.width**2 + root.height**2) /
    13**4 / 2
626             multiline: False
627             pos_hint: {"x": 0.2, "y":0.55}
628             size_hint: 0.2, 0.1
629             hint_text: "Decimal Degrees"
630 
631         TextInput:
632             id: dec
633             font_size: (root.width**2 + root.height**2) /
    13**4 / 2
634             multiline: False
635             pos_hint: {"x": 0.42, "y":0.55}
636             size_hint: 0.2, 0.1
637             hint_text: "Decimal Degrees"
638 
639         Label:
640             pos_hint:{"x": 0.1, "y":0.4}
641             size_hint: 0.05, 0.1
642             text: "Uncertainties \n(\u03B1, \u03B4): "
643 
644         TextInput:
645             id: sigma_ra
646             font_size: (root.width**2 + root.height**2) /
    13**4 / 2
647             multiline: False
648             pos_hint: {"x": 0.2, "y":0.4}
649             size_hint: 0.2, 0.1
650             hint_text: "Decimal Degrees"
651 
652         TextInput:
653             id: sigma_dec
654             font_size: (root.width**2 + root.height**2) /
    13**4 / 2
655             multiline: False
656             pos_hint: {"x": 0.42, "y":0.4}
657             size_hint: 0.2, 0.1
658             hint_text: "Decimal Degrees"
659 
660         Label:
661             text_size: self.size
662             halign: 'left'
```

```
663             pos_hint:{"x": .05, "y":0.25}
664             size_hint:0.2, 0.2
665             text: "Observations:    "
666 
667         ScrollView:
668             do_scroll_x: True
669             do_scroll_y: True
670             pos_hint:{"x": .2, "y":0.1}
671             size_hint:0.6, 0.2
672             border_color: 'red'
673             Label:
674                 halign: 'left'
675                 id: observation_output
676                 size_hint_y: None
677 
678         Button:
679             pos_hint:{"x":0.75, "y": .4}
680             size_hint:0.2,0.1
681             text: "Return to main page"
682             on_release:
683                 app.root.current = "main"
684                 root.manager.transition.direction = "right
    "
685 
686         Button:
687             pos_hint:{"x":0.75, "y": .6}
688             size_hint:0.2,0.1
689             text: "Clear Observations"
690             color: 1, 0, 0, 1
691             on_release:
692                 root.clear_observations()
693 
694         Button:
695             pos_hint:{"x":0.75, "y": .8}
696             size_hint:0.2,0.1
697             text: "Add Observation"
698             on_release:
699                 root.add_observation()
700 
701         Button:
702             pos_hint:{"x":0.75, "y": .0}
703             size_hint:0.2,0.1
704             text: "DEMO ONLY"
705             on_release:
706                 root.demo_only()
707 
```

```
708 <ResultsScreen>:
709     output_text: output_text
710 
711     FloatLayout:
712         Label:
713             pos_hint:{"x": 0.1, "y":0.8}
714             size_hint: 0.1, 0.1
715             text: "Output: "
716 
717         ScrollView:
718             id: scrlv
719             do_scroll_x: True
720             do_scroll_y: True
721             pos_hint:{"x": .2, "top":0.85}
722             size_hint:0.7, 0.6
723             border_color: 'red'
724             TextInput:
725                 id: output_text
726                 height: max(self.minimum_height, scrlv.
    height)
727                 font_size: (root.width**2 + root.height**2
    ) / 13**4 / 3
728                 multiline: True
729 
730         Button:
731             pos_hint:{"x": 0.2, "y":0.1}
732             size_hint: 0.28, 0.1
733             text: "Return to main page"
734             on_release:
735                 app.root.current = "main"
736                 root.manager.transition.direction = "down"
737 
738         Button:
739             pos_hint:{"x": 0.52, "y":0.1}
740             size_hint: 0.28, 0.1
741             text: "Reset Field"
742             on_release:
743                 root.on_enter()
744 
745 
```

```
 1 from kivy.app import App
 2 from kivy.lang import Builder
 3 from kivy.uix.screenmanager import ScreenManager, Screen
 4 from kivy.properties import ObjectProperty
 5 from kivy.uix.popup import Popup
 6 from kivy.uix.label import Label
 7 from src.dto import Observation
 8 from astropy.time import Time
 9 import numpy as np
10 from src.interface.tle_dto import TLE
11 from src.dto import PropParams, FilterOutput
12 from src.perturbation_util import *
13 from src.enums import Perturbations, Frames, Angles
14 from src.interface.string_conversions import dms_to_dd
15 from src.interface.cleaning import
   convert_obs_from_lla_to_eci
16 from src.core import milani
17 from verification.util import get_period, build_epochs,
   build_observations
18
19 tag_string = "Developed by Austin Ogle  Ver. 0.0.1  7/20/
   2020"
20
21
22 class MainWindow(Screen):
23     state = np.zeros(6)
24     epoch = Time("2000-01-01T00:00:00.000", format='isot',
   scale='utc')
25     prop_params = PropParams(epoch)
26     delta_x_apr = np.zeros(6)
27     p_apr = np.zeros((6, 6))
28     observations = []
29
30     tag = ObjectProperty(None)
31
32     def on_enter(self, *args):
33         self.tag.text = tag_string
34         self.state_label.text = "Satellite state:    " +
   str(self.state)
35         self.epoch_label.text = "Epoch: " + self.epoch.fits
36         self.prop_params.epoch = self.epoch
37         self.params_label.text = "Included Perturbations: "
    + self.prop_params.tostring()
38         self.delta_x_apr_label.text = "A priori \u0394 x
   :    " + str(self.delta_x_apr)
39         self.p_apr_label.text = "A priori covariance:    "
```

```
40          self.p_apr_value.text = str(self.p_apr)
41          self.observations_output.text = ""
42          for observation in self.observations:
43              self.observations_output.text += observation.
   tostring() + "\n"
44
45      def set_default_values(self):
46          self.state = np.zeros(6)
47          self.epoch = Time("2000-01-01T00:00:00.000", format
   ='isot', scale='utc')
48          self.prop_params = PropParams(self.epoch)
49          self.delta_x_apr = np.zeros(6)
50          self.p_apr = np.zeros((6, 6))
51          self.observations = []
52          self.on_enter()
53
54      def run_clicked(self):
55          if self.validate_values():
56              a_priori = FilterOutput(delta_x=self.
   delta_x_apr, p=self.p_apr)
57              sm.get_screen("results").output = milani(self.
   state, self.observations, self.prop_params, a_priori)
58              sm.current = 'results'
59
60      def validate_values(self):
61          if np.array_equal(self.state, np.zeros(6)):
62              invalid_entry("Input state vector")
63              return False
64          if self.epoch == Time("2000-01-01T00:00:00.000",
   format='isot', scale='utc'):
65              invalid_entry("Input epoch")
66              return False
67          if self.observations is []:
68              invalid_entry("Input observations")
69              return False
70          return True
71
72  class AddCoreValues(Screen):
73      state = ObjectProperty(None)
74      tle = ObjectProperty(None)
75      is_isot = ObjectProperty(None)
76      is_jd = ObjectProperty(None)
77      tle_active = ObjectProperty(None)
78      include_j2 = ObjectProperty(None)
79      include_j3 = ObjectProperty(None)
80      include_solar = ObjectProperty(None)
```

```
 81      include_lunar = ObjectProperty(None)
 82      include_srp = ObjectProperty(None)
 83      include_drag = ObjectProperty(None)
 84
 85      srp_A = ObjectProperty(None)
 86      srp_C_r = ObjectProperty(None)
 87      srp_m = ObjectProperty(None)
 88      drag_A = ObjectProperty(None)
 89      drag_C_d = ObjectProperty(None)
 90      drag_m = ObjectProperty(None)
 91
 92      format = "isot"
 93
 94      def on_enter(self, *args):
 95          self.epoch.hint_text = "YYYY-MM-DDTHH:MM:SS.SSS"
 96          self.is_isot.active = True
 97          self.format = 'isot'
 98          self.tle.readonly = True
 99
100      def j2_active(self, state):
101          if state:
102              sm.get_screen('main').prop_params.
    add_perturbation(Perturbations.J2, build_j2())
103          else:
104              del sm.get_screen('main').prop_params.
    perturbations[Perturbations.J2]
105
106      def j3_active(self, state):
107          if state:
108              sm.get_screen('main').prop_params.
    add_perturbation(Perturbations.J3, build_j3())
109          else:
110              del sm.get_screen('main').prop_params.
    perturbations[Perturbations.J3]
111
112      def solar_active(self, active):
113          if active:
114              if self.validate_values() is True:
115                  epoch = self.get_epoch_from_box()
116                  sm.get_screen('main').prop_params.
    add_perturbation(Perturbations.Sun, build_solar_third_body
    (epoch))
117              else:
118                  invalid_entry("Ensure epoch meets expected
     format")
119                  self.include_solar.active = False
```

```
120         else:
121             try:
122                 del sm.get_screen('main').prop_params.
    perturbations[Perturbations.Sun]
123             except KeyError:
124                 thing = "Hih"
125 
126     def lunar_active(self, active):
127         if active:
128             if self.validate_values() is True:
129                 epoch = self.get_epoch_from_box()
130                 sm.get_screen('main').prop_params.
    add_perturbation(Perturbations.Moon,
    build_lunar_third_body(epoch))
131             else:
132                 invalid_entry("Ensure epoch meets expected
     format")
133                 self.include_lunar.active = False
134         else:
135             try:
136                 del sm.get_screen('main').prop_params.
    perturbations[Perturbations.Moon]
137             except KeyError:
138                 thing = "Hih"
139 
140     def drag_active(self, active):
141         if active:
142             self.drag_A.readonly = False
143             self.drag_A.background_color = 1, 1, 1, 1
144             self.drag_A.hint_text_color = .6, .6, .6, 1
145             self.drag_C_d.readonly = False
146             self.drag_C_d.background_color = 1, 1, 1, 1
147             self.drag_C_d.hint_text_color = .6, .6, .6, 1
148             self.drag_m.readonly = False
149             self.drag_m.background_color = 1, 1, 1, 1
150             self.drag_m.hint_text_color = .6, .6, .6, 1
151         else:
152             self.drag_A.readonly = True
153             self.drag_A.background_color = .6, .6, .6, 1
154             self.drag_A.hint_text_color = 1, 1, 1, 1
155             self.drag_C_d.readonly = True
156             self.drag_C_d.background_color = .6, .6, .6, 1
157             self.drag_C_d.hint_text_color = 1, 1, 1, 1
158             self.drag_m.readonly = True
159             self.drag_m.background_color = .6, .6, .6, 1
160             self.drag_m.hint_text_color = 1, 1, 1, 1
```

```
161             try:
162                 del sm.get_screen('main').prop_params.
    perturbations[Perturbations.Drag]
163             except KeyError:
164                 thing = "Hih"
165 
166     def srp_active(self, active):
167         if active:
168             self.srp_A.readonly = False
169             self.srp_A.background_color = 1, 1, 1, 1
170             self.srp_A.hint_text_color = .6, .6, .6, 1
171             self.srp_C_r.readonly = False
172             self.srp_C_r.background_color = 1, 1, 1, 1
173             self.srp_C_r.hint_text_color = .6, .6, .6, 1
174             self.srp_m.readonly = False
175             self.srp_m.background_color = 1, 1, 1, 1
176             self.srp_m.hint_text_color = .6, .6, .6, 1
177         else:
178             self.srp_A.readonly = True
179             self.srp_A.background_color = .6, .6, .6, 1
180             self.srp_A.hint_text_color = 1, 1, 1, 1
181             self.srp_C_r.readonly = True
182             self.srp_C_r.background_color = .6, .6, .6, 1
183             self.srp_C_r.hint_text_color = 1, 1, 1, 1
184             self.srp_m.readonly = True
185             self.srp_m.background_color = .6, .6, .6, 1
186             self.srp_m.hint_text_color = 1, 1, 1, 1
187             try:
188                 del sm.get_screen('main').prop_params.
    perturbations[Perturbations.SRP]
189             except KeyError:
190                 thing = "Hih"
191 
192     def get_epoch_from_box(self):
193         if self.format == 'jd':
194             epoch = Time(float(self.epoch.text), format=
    self.format, scale='utc')
195         elif self.format == 'isot':
196             epoch = Time(self.epoch.text, format=self.
    format, scale='utc')
197         return epoch
198 
199     def tle_state(self, state):
200         if state:
201             self.tle.readonly = False
202             self.tle.background_color = [1, 1, 1, 1]
```

```
203             self.tle.hint_text = ""
204             self.state.background_color = [.6, .6, .6, 1]
205             self.epoch.background_color = [.6, .6, .6, 1]
206             self.state.readonly = True
207             self.epoch.readonly = True
208 
209         else:
210             self.tle.readonly = True
211             self.tle.background_color = [.6, .6, .6, 1]
212             self.tle.hint_text = "This box is read only
    until the checkbox has been activated"
213             self.tle.hint_text_color = [1, 1, 1, 1]
214             self.state.background_color = [1, 1, 1, 1]
215             self.epoch.background_color = [1, 1, 1, 1]
216             self.state.readonly = False
217             self.epoch.readonly = False
218 
219     def isot_active(self):
220         self.epoch.hint_text = "YYYY-MM-DDTHH:MM:SS.SSS"
221         self.format = 'isot'
222 
223     def jd_active(self):
224         self.epoch.hint_text = "2000000.000"
225         self.format = 'jd'
226 
227     def update_values(self):
228         if self.validate_values():
229             if self.include_srp.active:
230                 sm.get_screen("main").prop_params.
    add_perturbation(Perturbations.SRP,
231 
           build_srp(float(self.srp_C_r.text),
232 
                     float(self.srp_A.text),
233 
                     float(self.srp_m.text),
234 
                     self.get_epoch_from_box()))
235             if self.include_drag.active:
236                 sm.get_screen("main").prop_params.
    add_perturbation(Perturbations.Drag,
237 
           build_basic_drag(float(self.drag_C_d.text),
238 
                            float(self.drag_A.text),
239 
```

```
239                                 float(self.drag_m.text)))
240             if self.tle_active.active is False:
241                 sm.get_screen('main').state = np.
    fromstring(self.state.text, sep=",").reshape(6)
242                 sm.get_screen('main').epoch = self.
    get_epoch_from_box()
243             else:
244                 tle = TLE.from_lines(self.tle.text)
245                 sm.get_screen('main').state, sm.get_screen
    ('main').epoch = tle.to_state()
246             sm.current = 'main'
247 
248     def validate_values(self):
249         if self.tle_active.active is False:
250             try:
251                 np.fromstring(self.state.text, sep=",").
    reshape(6)
252             except ValueError:
253                 invalid_entry("The state entry is invalid
    . Separate entries by commas.")
254                 return False
255             try:
256                 if self.format == 'jd':
257                     Time(float(self.epoch.text), format=
    self.format, scale='utc')
258                 elif self.format == 'isot':
259                     Time(self.epoch.text, format=self.
    format, scale='utc')
260             except ValueError:
261                 invalid_entry("The epoch format is
    incorrect. See hint")
262                 return False
263             return True
264         else:
265             try:
266                 TLE.from_lines(self.tle.text)
267             except ValueError:
268                 invalid_entry("The TLE format is incorrect
    .")
269                 return False
270             return True
271 
272 
273 class AddAPrioriValues(Screen):
274     delta_x_apr = ObjectProperty(None)
275     p_apr = ObjectProperty(None)
```

```
276     state = ObjectProperty(None)
277
278     def update_values(self):
279         if self.state.text == "" and self.p_apr.text == ""
    :
280             sm.current = 'main'
281         elif self.validate_values() is True:
282             sm.get_screen('main').delta_x_apr = np.
    fromstring(self.state.text, sep=",")
283             p_lines = self.p_apr.text.replace("\n", ",")
284             p = np.fromstring(p_lines, sep=",").reshape((6
    , 6))
285             sm.get_screen('main').p_apr = p
286             sm.current = "main"
287
288     def clear_values(self):
289         sm.get_screen('main').delta_x_apr = np.zeros(6)
290         sm.get_screen('main').p_apr = np.zeros((6, 6))
291         self.p_apr.text = ""
292         self.state.text = ""
293
294     def validate_values(self):
295         if self.state.text == "" and self.p_apr.text == ""
    :
296             return True
297         try:
298             p_lines = self.p_apr.text.replace("\n", ",")
299             p = np.fromstring(p_lines, sep=",").reshape((6
    , 6))
300         except ValueError:
301             invalid_entry("Covariance Matrix input didn't
    match \nthe expected format. See hint")
302             return False
303         try:
304             np.fromstring(self.state.text, sep=",").
    reshape((6, 1))
305         except ValueError:
306             invalid_entry("\u0394 x (a priori) didn't
    match the expected format. See hint")
307             return False
308         return True
309
310
311 class AddObservation(Screen):
312     new_string = ObjectProperty(None)
313     epoch = ObjectProperty(None)
```

```
314     ra = ObjectProperty(None)
315     dec = ObjectProperty(None)
316     sigma_ra = ObjectProperty(None)
317     sigma_dec = ObjectProperty(None)
318     observation_output = ObjectProperty(None)
319     is_isot = ObjectProperty(None)
320     is_jd = ObjectProperty(None)
321     lat = ObjectProperty(None)
322     lon = ObjectProperty(None)
323     alt = ObjectProperty(None)
324     is_dd = ObjectProperty(None)
325     is_dms = ObjectProperty(None)
326 
327     epoch_format = 'isot'
328     location_format = "dd"
329 
330     def on_enter(self, *args):
331         self.epoch.hint_text = "YYYY-MM-DDTHH:MM:SS.SSS"
332         self.is_isot.active = True
333         self.epoch_format = 'isot'
334         self.is_dd.active = True
335 
336     def add_observation(self):
337         if self.validate_values() is True:
338             obs_pos = self.get_obs_pos()
339             obs = Observation(obs_pos, None, self.epoch.
    text, np.array([float(self.ra.text), float(self.dec.text
    )]),
340                               None, np.array([float(self.
    sigma_ra.text), float(self.sigma_dec.text)]))
341             sm.get_screen('main').observations.append(obs)
342             self.clear_values()
343             self.observation_output.text = ""
344             for observation in sm.get_screen('main').
    observations:
345                 self.observation_output.text +=
    observation.tostring() + "\n"
346 
347     def clear_observations(self):
348         sm.get_screen('main').observations = []
349         self.observation_output.text = ""
350         sm.get_screen('main').observations_output.text =
    ""
351 
352     def clear_values(self):
353         self.lat.text = ""
```

```
354         self.lon.text = ""
355         self.alt.text = ""
356         self.epoch.text = ""
357         self.ra.text = ""
358         self.dec.text = ""
359         self.sigma_ra.text = ""
360         self.sigma_dec.text = ""
361 
362     def validate_values(self):
363         try:
364             float(self.ra.text)
365             float(self.dec.text)
366             float(self.sigma_dec.text)
367             float(self.sigma_ra.text)
368         except ValueError:
369             invalid_entry("Invalid entries for
    measurements and their uncertainties.\n D"
370                           "ecimal degrees is the expected
    format ")
371             return False
372         try:
373             if self.epoch_format == 'jd':
374                 sm.get_screen('main').epoch = Time(float(
    self.epoch.text), format=self.epoch_format, scale='utc')
375             elif self.epoch_format == 'isot':
376                 sm.get_screen('main').epoch = Time(self.
    epoch.text, format=self.epoch_format, scale='utc')
377         except ValueError:
378             invalid_entry("The epoch format is incorrect.
    See hint")
379             return False
380         try:
381             if self.is_dd.active is True:
382                 float(self.lat.text)
383                 float(self.lon.text)
384             else:
385                 dms_to_dd(self.lat.text)
386                 dms_to_dd(self.lon.text)
387         except ValueError:
388             invalid_entry("The lat/lon inputs do not match
     expected format")
389             return False
390         try:
391             float(self.alt.text)
392         except ValueError:
393             invalid_entry("The alt input does not match
```

```
393 expected format")
394             return False
395         return True
396 
397     def get_obs_pos(self):
398         if self.is_dd.active is True:
399             lat = float(self.lat.text)
400             lon = float(self.lon.text)
401         else:
402             lat = dms_to_dd(self.lat.text)
403             lon = dms_to_dd(self.lon.text)
404         return [lat * u.deg, lon * u.deg, float(self.alt.
    text) * u.km]
405 
406     def isot_active(self):
407         self.epoch.hint_text = "YYYY-MM-DDTHH:MM:SS.SSS"
408         self.epoch_format = 'isot'
409 
410     def jd_active(self):
411         self.epoch.hint_text = "2000000.000"
412         self.epoch_format = 'jd'
413 
414     def dd_active(self):
415         self.lat.hint_text = "00.000"
416         self.lon.hint_text = "00.000"
417 
418     def dms_active(self):
419         self.lat.hint_text = "00 00\' 00\""
420         self.lon.hint_text = "00 00\' 00\""
421 
422     def demo_only(self):
423         x = np.array([5748.6001, 2679, 3443, 4.33, -1.922
    , -5.726])
424         x_offset = np.array([500, 100, 100, .2, .1, .1])
425         x_true = x + x_offset
426         period = get_period(x)
427         epoch = Time(2454283.0, format="jd", scale="tdb")
428         obs_pos = [29.2108 * u.deg, 81.0228 * u.deg, 3.
    9624 * u.km]  # Daytona Beach, except 13 feet above sea
    level
429         prop_params = PropParams(epoch)
430         step = period / 32 * u.s
431         epochs = build_epochs(epoch, step, 5)
432         observations = build_observations(x_true,
    prop_params, obs_pos, Frames.LLA, epochs)
433         self.observation_output.text = ""
```

```
434         for observation in observations:
435             self.observation_output.text += observation.
    tostring() + "\n"
436         sm.get_screen("main").observations = observations
437 
438 
439 class ResultsScreen(Screen):
440     output_text = ObjectProperty(None)
441     output = FilterOutput()
442 
443     def on_enter(self, *args):
444         self.output_text.text = self.output.tostring()
445         if sm.get_screen('addcore').tle_active.active is
    True:
446             updated_tle = TLE.from_lines(sm.get_screen("
    addcore").tle.text).update(self.output.x_out, self.output.
    epoch)
447             self.output_text.text += "\n\n" + updated_tle.
    to_string()
448 
449 
450 def invalid_entry(string):
451     pop = Popup(title='Invalid Entry',
452                 content=Label(text=string),
453                 size_hint=(None, None), size=(400, 400))
454 
455     pop.open()
456 
457 
458 class WindowManager(ScreenManager):
459     pass
460 
461 
462 kv = Builder.load_file("astro.kv")
463 
464 sm = WindowManager()
465 
466 screens = [MainWindow(name="main"), AddObservation(name="
    obs"), AddAPrioriValues(name='apr'),
467            AddCoreValues(name='addcore'), ResultsScreen(
    name='results')]
468 for screen in screens:
469     sm.add_widget(screen)
470 
471 sm.current = "main"
472 
```

```
473
474 class BatchLeastSquaresFilterApp(App):
475     def build(self):
476         return sm
477
478
479 if __name__ == "__main__":
480     BatchLeastSquaresFilterApp().run()
```

9.2 Verification

```
 1 from astropy.time import Time
 2 import astropy.units as u
 3 from src.enums import Frames
 4 from src.observation_function import y
 5 from src.state_propagator import state_propagate
 6 from src.dto import PropParams, Observation
 7 from src.interface.cleaning import
   convert_obs_from_lla_to_eci
 8 from src.interface.local_angles import
   get_local_angles_via_state_propagation
 9 from src.interface.tle_dto import TLE
10
11 obs_pos = [29.218103 * u.deg, -81.031723 * u.deg, 0 * u.km]
12 tle_string = """
13 ISS (ZARYA)
14 1 25544U 98067A   20211.19695584  .00000552  00000-0  17878
   -4 0  9990
15 2 25544  51.6419 140.5349 0000882 143.5591 191.9640 15.
   49512638238524
16 """
17 tle = TLE.from_lines(tle_string)
18 x, epoch_i = tle.to_state()
19
20 epoch_1 = Time("2020-7-25T00:00:00.000")
21 epoch_2 = Time("2020-7-25T00:05:00.000")
22 epoch_3 = Time("2020-7-25T00:10:00.000")
23 epoch_4 = Time("2020-7-25T00:15:00.000")
24
25 x_1 = state_propagate(x, epoch_i, PropParams(epoch_1))
26 x_2 = state_propagate(x, epoch_i, PropParams(epoch_2))
27 x_3 = state_propagate(x, epoch_i, PropParams(epoch_3))
28 x_4 = state_propagate(x, epoch_i, PropParams(epoch_4))
29
30 observation_1 = convert_obs_from_lla_to_eci(Observation(
   obs_pos, Frames.LLA, epoch_1, None, None, None))
31 observation_2 = convert_obs_from_lla_to_eci(Observation(
   obs_pos, Frames.LLA, epoch_2, None, None, None))
32 observation_3 = convert_obs_from_lla_to_eci(Observation(
   obs_pos, Frames.LLA, epoch_3, None, None, None))
33 observation_4 = convert_obs_from_lla_to_eci(Observation(
   obs_pos, Frames.LLA, epoch_4, None, None, None))
34
35 print(y(x_1, observation_1))
36 print(y(x_2, observation_2))
37 print(y(x_3, observation_3))
38 print(y(x_4, observation_4))
```

```
39
40
```

```python
import numpy as np
import math
from src.dto import PropParams, Observation
from src.enums import Angles, Frames
from src.state_propagator import state_propagate
from src.frames import lla_to_ecef, ecef_to_eci
from src.observation_function import y
from src.constants import mu
import astropy.units as u


def generate_earth_surface():
    """
    Generates x,y,z coordinates for a perfect sphere representing the Earth. To be used in plot_surface()
    """
    r = 6378
    u = np.linspace(0, 2 * np.pi, 50)
    v = np.linspace(0, np.pi, 50)
    x = r * np.outer(np.cos(u), np.sin(v))
    y = r * np.outer(np.sin(u), np.sin(v))
    z = r * np.outer(np.ones(np.size(u)), np.cos(v))
    return x, y, z


def get_a(x):
    """
    Returns semi-major axis of an orbit given the state x = [r v]. Unit: [km]
    """
    rr = x[0:3]
    vv = x[3:6]
    v = np.linalg.norm(vv)
    r = np.linalg.norm(rr)
    eps = v*v/2 - (mu.value/r)
    a = -mu.value/(2*eps)
    return a


def get_period(x):
    """
    Returns the period of an orbit given the state x = [r v]. Unit: [s]
    """
    a = get_a(x)
    t = 2*np.pi*math.sqrt(a*a*a/mu.value)
```

```
44      return t
45
46
47  def get_e(x):
48      """
49      Returns the eccentricity of an orbit given the state x
     = [r v]
50      """
51      rr = x[0:3]
52      vv = x[3:6]
53      r = np.linalg.norm(rr)
54      hh = np.cross(rr, vv)
55      ee = np.cross(vv/mu, hh) - (rr/r)
56      e = np.linalg.norm(ee)
57      return e
58
59
60  def get_satellite_position_over_time(x, init_epoch, epochs
    ):
61      r = np.zeros((len(epochs), 3))
62      r[0] = x[0:3]
63      params = PropParams(init_epoch)
64      for i in range(1, len(epochs)):
65          x_temp = state_propagate(x, epochs[i], params)
66          r[i] = x_temp[0:3]
67      return r, epochs
68
69
70  sigma_theta = .003
71
72
73  def build_observations(x, prop_params, obs_pos, frame,
    epochs, sigmas=np.array([sigma_theta, sigma_theta])):
74      output = []
75      temp = []
76      assert frame == Frames.LLA
77      for k in range(len(epochs)):
78          epoch = epochs[k]
79          x_k = state_propagate(x, epoch, prop_params)
80          pos = ecef_to_eci(lla_to_ecef(obs_pos), epoch)
81          temp.append(Observation(pos, Frames.ECI, epoch,
    None, Angles.Local, sigmas))
82          obs_values = y(x_k, temp[k])
83          output.append(Observation(pos, Frames.ECI, epoch,
    obs_values, Angles.Local, sigmas))
84      return output
```

```
85
86
87 def build_epochs(epoch, stepsize, steps):
88     epochs = []
89     for i in range(steps):
90         epochs.append(epoch + i * stepsize)
91     return epochs
92
93
94 def build_noisy_observations(x, prop_params, obs_pos,
   frame, epochs, noise=1/60):
95     observations = build_observations(x, prop_params,
   obs_pos, frame, epochs, sigmas=np.array([noise, noise]))
96     for obs in observations:
97         obs.obs_values = obs.obs_values + np.random.rand(2
   ) * noise
98     return observations
99
```

```
 1 from astropy.time import Time
 2 from astropy.coordinates import solar_system_ephemeris,
   get_body_barycentric
 3 import astropy.units as u
 4 from verification.util import build_observations,
   build_epochs, get_satellite_position_over_time
 5 from src.interface.local_angles import local_angles
 6 from src.frames import eci_to_icrs, lla_to_ecef,
   ecef_to_eci, icrs_to_eci, eci_to_ecef
 7 from src.observation_function import get_ra_and_dec
 8
 9 obs_pos = [29.218103 * u.deg, -81.031723 * u.deg, 0 * u.km]
10
11 desired_epoch = Time("2020-07-16T08:00:00.000", format="
   isot", scale="tdb")
12 dt = 1
13 tf = 14 * dt * u.h
14 epochs = build_epochs(desired_epoch, dt * u.h, round((tf/dt
   ).value))
15
16 for i in range(len(epochs)):
17     # print(epochs[i])
18     obs = eci_to_icrs(ecef_to_eci(lla_to_ecef(obs_pos),
   epochs[1]), epochs[i])
19     venus = get_body_barycentric('venus', epochs[i]).xyz.to
   (u.km).value
20     rr = venus - obs
21     # print(get_ra_and_dec(rr))
22     rr_ecef = eci_to_ecef(icrs_to_eci(rr, epochs[i]),
   epochs[i])
23     thing = local_angles(rr_ecef, obs_pos)
24     print(thing)
25
26
27  # Date__(UT)__HR:MN     R.A.___(ICRF)___DEC R.A._(a-appar)
   _DEC. Azi_(a-appr)_Elev
28  # 2020-Jul-16 08:00  m   71.39000  17.78207  71.67881  17.
   81745  70.5231   1.7878
29  # 2020-Jul-16 09:00  m   71.41659  17.78600  71.70544  17.
   82133  77.3749  14.3731
30  # 2020-Jul-16 10:00 Nm   71.44291  17.78992  71.73179  17.
   82521  83.9301  27.2924
31  # 2020-Jul-16 11:00 *m   71.46897  17.79381  71.75789  17.
   82906  90.9000  40.3707
32  # 2020-Jul-16 12:00 *m   71.49483  17.79766  71.78378  17.
   83286  99.5451  53.4093
```

```
33  # 2020-Jul-16 13:00 *m   71.52053  17.80144  71.80951  17.
   83660 113.2688  65.9919
34  # 2020-Jul-16 14:00 *m   71.54614  17.80514  71.83514  17.
   84026 145.2445  76.3856
35  # 2020-Jul-16 15:00 *m   71.57175  17.80876  71.86077  17.
   84384 210.4835  76.9496
36  # 2020-Jul-16 16:00 *m   71.59743  17.81229  71.88646  17.
   84733 245.1981  66.9476
37  # 2020-Jul-16 17:00 *m   71.62327  17.81575  71.91230  17.
   85074 259.6529  54.4415
38  # 2020-Jul-16 18:00 *m   71.64932  17.81914  71.93836  17.
   85409 268.5288  41.4208
39  # 2020-Jul-16 19:00 *m   71.67566  17.82248  71.96469  17.
   85739 275.5757  28.3407
40  # 2020-Jul-16 20:00 *m   71.70232  17.82579  71.99134  17.
   86065 282.1376  15.4071
41  # 2020-Jul-16 21:00 *m   71.72932  17.82910  72.01834  17.
   86391 288.9475   2.7941
```

```
 1 import numpy as np
 2 import scipy.linalg as la
 3 from astropy.time import Time
 4 from astropy.coordinates import solar_system_ephemeris,
   get_body_barycentric
 5 import astropy.units as u
 6 from verification.util import build_observations,
   build_epochs, get_satellite_position_over_time
 7 from src.interface.tle_dto import TLE
 8 from src.enums import Frames
 9 from src.dto import PropParams
10 from src.interface.local_angles import
   get_local_angles_via_state_propagation
11 from src.frames import eci_to_icrs, lla_to_ecef,
   ecef_to_eci, eci_to_angles
12 from src.observation_function import get_ra_and_dec
13
14
15 tle_string = """
16 ISS (ZARYA)
17 1 25544U 98067A   20202.44882139 -.00000250  00000-0  35814
   -5 0  9999
18 2 25544  51.6421 183.8397 0001306 134.1760 329.9627 15.
   49516142237178
19 """
20 obs_pos = [29.218103 * u.deg, -81.031723 * u.deg, 0 * u.km]
21 tle = TLE.from_lines(tle_string)
22
23 x, epoch = tle.to_state()
24 params = PropParams(epoch)
25 desired_epoch = Time("2020-07-20T00:00:00.000", format="
   isot", scale="utc")
26 dt = 1
27 epochs = build_epochs(desired_epoch, dt * u.h, 48)
28
29 obj_eci, epochs = get_satellite_position_over_time(x, epoch
   , epochs)
30 for i in range(len(epochs)):
31     # print(epochs[i])
32     print(eci_to_angles(obj_eci[i], epochs[i]))
33     # obj = eci_to_icrs(obj_eci[i], epochs[i])
34     # obs = eci_to_icrs(ecef_to_eci(lla_to_ecef(obs_pos),
   epochs[1]), epochs[i])
35     # rr = obj - obs
36     # print(rr)
37     # print(get_ra_and_dec(rr))
```

```
38      # print(rr)
39      # print(positions[i])
40      # print("norms")
41      # print(la.norm(positions[i]))
42
43  # observations = build_observations(x, params, obs_pos,
    Frames.LLA, epochs)
44
45  # for obs in observations:
46  #     print(obs.obs_values)
47  #
48  # n = len(epochs)
49  # locals = get_local_angles_via_state_propagation(x, params
    , epochs[0], epochs[n-1], n-2, obs_pos, Frames.LLA)
50  # for local in locals:
51  #     local[2].format = 'isot'
52  #     print(local)
53  print(epochs)
54   # Date__(UT)__HR:MN     R.A.___(ICRF)___DEC R.A._(a-appar)
    _DEC. Azi_(a-appr)_Elev
55   # 2020-Jul-20 00:00 *     0.05070  10.84112   0.31231  10.
    95381  47.0208 -36.3402
56   # 2020-Jul-20 01:00 N   127.30069 -35.33755 127.48970 -35.
    40576 242.8426 -27.6417
57   # 2020-Jul-20 02:00      51.59770 -32.11709  51.79858 -32.
    04375 105.8656 -76.3935
58   # 2020-Jul-20 03:00      36.35511  42.44152  36.67421  42.
    52912  32.0578  -6.8717
59   # 2020-Jul-20 04:00     133.00907 -30.65197 133.21326 -30.
    72887 258.1257 -59.0407
60   # 2020-Jul-20 05:00      60.83152 -31.49589  61.02673 -31.
    43788 106.3262 -46.0039
61   # 2020-Jul-20 06:00     177.09639  22.66602 177.35492  22.
    55637 311.5023 -19.3800
62   # 2020-Jul-20 07:00     143.49001 -34.02932 143.69799 -34.
    12051 191.5524 -84.9907
63   # 2020-Jul-20 08:00      57.96989 -44.78454  58.13028 -44.
    72084 133.7371 -14.0170
64   # 2020-Jul-20 09:00     196.79994   6.45559 197.05449   6.
    34949 322.5631 -47.3431
65   # 2020-Jul-20 10:00 N   156.62333 -45.15268 156.83166 -45.
    25790 132.0319 -60.9900
66   # 2020-Jul-20 11:00 *m  289.71007 -16.31613 290.00593 -16.
    27707 256.9086 -10.1073
67   # 2020-Jul-20 12:00 *m  209.70439  -2.12347 209.96598  -2.
    22034  16.6401 -62.0193
```

```
68  # 2020-Jul-20 13:00 *m  183.97100 -71.82888 184.24794 -71
   .94567 162.3539 -40.0443
69  # 2020-Jul-20 14:00 *m  289.69545   4.51585 289.95167   4
   .55514 302.7144 -37.2994
70  # 2020-Jul-20 15:00 *m  220.67798 -13.70392 220.95654 -13
   .78940  77.3843 -48.3860
71  # 2020-Jul-20 16:00 *m  336.46718 -60.94432 336.81519 -60
   .83737 213.1399 -40.2083
72  # 2020-Jul-20 17:00 *m  298.90320  10.03496 299.15043  10
   .09059 350.3511 -50.2313
73  # 2020-Jul-20 18:00 *m  228.02345 -42.82220 228.36331 -42
   .89995 126.3097 -26.6128
74  # 2020-Jul-20 19:00 *m  356.44049 -33.36608 356.70736 -33
   .25017 253.3324 -61.0073
75  # 2020-Jul-20 20:00 *m  310.17876  14.99673 310.42037  15
   .07025  34.7612 -38.3106
76  # 2020-Jul-20 21:00 *m   89.23729 -74.86550  89.06153 -74
   .86114 196.8826 -26.5753
77  # 2020-Jul-20 22:00 *m    9.66365 -20.51236   9.91737 -20
   .39852 349.0339 -81.0286
78  # 2020-Jul-20 23:00 *m  326.25103  29.70246 326.48005  29
   .79578  47.4093 -10.0004
79  # 2020-Jul-21 00:00 *m   83.08374 -47.43661  83.21439 -47
   .42021 229.4728 -50.9677
80  # 2020-Jul-21 01:00 N    21.80298 -12.98255  22.05298 -12
   .87576  67.1481 -56.7973
81  # 2020-Jul-21 02:00     141.22941  12.28640 141.49877  12
   .19998 288.9919  -8.2196
82  # 2020-Jul-21 03:00      92.17045 -40.75729  92.32598 -40
   .75937 210.1181 -76.3150
83  # 2020-Jul-21 04:00      34.00058  -4.94728  34.25369  -4
   .85247  81.2064 -24.9104
84  # 2020-Jul-21 05:00     157.69971  -5.93901 157.95025  -6
   .04261 290.0032 -42.0517
85  # 2020-Jul-21 06:00     102.14299 -43.67203 102.28964 -43
   .69406 131.9237 -64.6483
86  # 2020-Jul-21 07:00     177.18778  77.63935 177.45811  77
   .53176 352.6808  18.6859
87  # 2020-Jul-21 08:00     170.23523 -13.63711 170.48642 -13
   .74815 321.8402 -70.9670
88  # 2020-Jul-21 09:00     110.56211 -60.04299 110.62723 -60
   .08144 145.4774 -38.4035
89  # 2020-Jul-21 10:00 N   233.21607  18.24682 233.44809  18
   .18211 311.8956 -25.8188
90  # 2020-Jul-21 11:00 *   182.73384 -23.49275 182.99298 -23
   .60625  77.9328 -70.7089
```

```
 91  # 2020-Jul-21 12:00 *m  316.19959 -71.12170 316.70141 -71
    .03809 200.7886 -24.9651
 92  # 2020-Jul-21 13:00 *m  244.52480  12.98467 244.76398  12
    .93844 344.2579 -46.5155
 93  # 2020-Jul-21 14:00 *m  197.06027 -42.94636 197.34918 -43
    .05683 123.1760 -46.9716
 94  # 2020-Jul-21 15:00 *m  320.08000 -26.72032 320.38111 -26
    .63184 258.5667 -40.4298
 95  # 2020-Jul-21 16:00 *m  254.57883   9.69059 254.82360   9
    .66252  34.4005 -44.7450
 96  # 2020-Jul-21 17:00 *m  249.02834 -81.27252 250.00066 -81
    .31580 172.6341 -35.2762
 97  # 2020-Jul-21 18:00 *m  330.85280  -9.60718 331.12585  -9
    .50709 308.9823 -61.3533
 98  # 2020-Jul-21 19:00 *m  261.33641   3.07885 261.59412   3
    .06336  72.8700 -22.2137
 99  # 2020-Jul-21 20:00 *m   23.29123 -56.47938  23.48382 -56
    .37083 216.0127 -48.7058
100  # 2020-Jul-21 21:00 *m  342.97595  -0.33835 343.23916  -0
    .22944  21.5047 -59.2303
101  # 2020-Jul-21 22:00 *m  179.50051 -54.94334 179.75431 -55
    .05935 185.3538   5.3415
102  # 2020-Jul-21 23:00 *m   38.40174 -39.79080  38.60130 -39
    .69831 224.7710 -74.1571
103  # 2020-Jul-22 00:00 *m  357.78718   9.98262 358.04844  10
    .09546  51.5020 -34.1742
```

```python
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np
from verification.util import generate_earth_surface, get_satellite_position_over_time, build_observations, build_epochs
from src.core import milani
from src.dto import PropParams
from src.enums import Frames
from astropy.time import Time
import astropy.units as u
from verification.util import get_period

r = [5748.6001, 2679, 3443]
v = [4.33, -1.922, -5.726]
x = np.array([r[0], r[1], r[2], v[0], v[1], v[2]])
x_offset = np.array([500, 100, 100, .2, .1, .1])
x_true = x + x_offset
period = get_period(x)
dt = period / 100
tf = period * 2
epoch = Time(2454283.0, format="jd", scale="tdb")

obs_pos = [29.2108 * u.deg, 81.0228 * u.deg, 3.9624 * u.km]     #Daytona Beach, except 13 feet above sea level
prop_params = PropParams(epoch)
step = period/32 * u.s
epochs = build_epochs(epoch, step, 5)
observations = build_observations(x_true, prop_params, obs_pos, Frames.LLA, epochs)
output = milani(x, observations, prop_params)
x_alg = output.x_out
p = output.p

print("x alg")
print(x_alg)
print("x true")
print(x_true)

print("State residual")
print(x_true - x_alg)
print("Uncertainty")
print(np.diag(p))
print("initial offset")
print(x_offset)

```

```
# r_init = get_satellite_position_over_time(x, epoch_obs, tf, dt)
# r_offset = get_satellite_position_over_time(x + x_offset, epoch_obs, tf, dt)
# r_alg = get_satellite_position_over_time(x_alg, epoch_obs, tf, dt)
#
# n = r_alg.shape[0]-1
# print(r_offset[n, 0])
#
# fig = plt.figure()
# ax = fig.gca(projection='3d')
#
# x, y, z = generate_earth_surface()
# ax.plot3D(r_init[:, 0], r_init[:, 1], r_init[:, 2], color='red', label='initial')
# ax.plot3D(r_offset[:, 0], r_offset[:, 1], r_offset[:, 2], color='blue', label='offset')
# ax.plot3D(r_alg[:, 0], r_alg[:, 1], r_alg[:, 2], color='green', label='algorithm')
# ax.plot3D([r_offset[n, 0]], [r_offset[n, 1]], [r_offset[n, 2]], color='blue', label='Final Location', marker='o')
# ax.plot_surface(x, y, z, color='b')
#
# Re = 6378
# dim = 6378 * 15
# ax.set_xlim([-dim, dim])
# ax.set_ylim([-dim, dim])
# ax.set_zlim([-dim, dim])
# ax.set_xlabel('x [km]')
# ax.set_ylabel('y [km]')
# ax.set_zlabel('z [km]')
# ax.legend()
# plt.show()
#
```

```
 1 # This file serves to demonstrate the accuracy of poliastro
   's method for a two body scenario with no perturbations.
 2 # Integrating force over time using the dopri8 integrator
   will be compared to the Lagrange/Gibbs method.
 3
 4 import numpy as np
 5 import matplotlib.pyplot as plt
 6 import scipy.linalg as la
 7 from verification.util import *
 8 from src.state_propagator import state_propagate
 9 from src.dto import PropParams
10 from astropy.time import Time
11 import astropy.units as u
12 from poliastro.twobody import Orbit
13 from poliastro.bodies import Earth
14 from poliastro.twobody.propagation import cowell
15
16
17 r = [66666, 0, 0]
18 v = [0, -2.644, 0]
19 x = np.array([r[0], r[1], r[2], v[0], v[1], v[2]])
20
21 a = get_a(x)
22 e = get_e(x)
23 period = get_period(x)
24 dt = period / 100
25
26 #   Lagrange Gibbs Construction
27 rr0 = r - np.zeros(len(r))
28 vv0 = v - np.zeros(len(r))
29 r0 = la.norm(rr0)
30 v0 = la.norm(vv0)
31 n = math.sqrt(mu.value/(a*a*a))
32
33 t = np.arange(0, period*10, dt)
34 F = np.zeros((len(t), 1))
35 G = np.zeros((len(t), 1))
36 r_lg = np.zeros((len(t), 3))
37 M = n*t
38
39 for i in range(0, len(M)):
40     E = M[i]
41     for j in range(0, 8):
42         E = E + (M[i] - E+e*np.sin(E))/(1-e*np.cos(E))
43     F[i] = 1-(a/r0)*(1-np.cos(E))
44     G[i] = t[i] + math.sqrt(a*a*a/mu.value)*(np.sin(E)-E)
```

```
45      r_lg[i] = F[i]*rr0 + G[i]*vv0
46
47
48  #   Poliastro construction
49  r_poli = np.zeros((len(t), 3))
50  epoch = Time(2454283.0, format="jd", scale="tdb")
51  prop_params = PropParams(epoch)
52  for i in range(0, len(t)):
53      r_poli[i] = x[0:3]
54      epoch = epoch + dt * u.s
55      x = state_propagate(x, epoch, prop_params)
56      prop_params.epoch = epoch
57
58  #   Difference calculation
59  r_diff = r_lg - r_poli
60  diff = np.zeros((len(t), 1))
61  for i in range(0, len(t)):
62      diff[i] = la.norm(r_diff[i])
63
64  #   Plots orbits on top of one another
65  x, y, z = generate_earth_surface()
66  fig = plt.figure()
67  ax = fig.gca(projection='3d')
68  ax.plot3D(r_lg[:, 0], r_lg[:, 1], r_lg[:, 2], 'grey')
69  ax.plot3D(r_poli[:, 0], r_poli[:, 1], r_poli[:, 2], 'red')
70  ax.plot_surface(x, y, z, color='b')
71  dim = 11*6378
72  ax.set_xlim([-dim, dim])
73  ax.set_ylim([-dim, dim])
74  ax.set_zlim([-dim, dim])
75  ax.set_xlabel('x')
76  ax.set_ylabel('y')
77  ax.set_zlabel('z')
78  plt.show()
79
80  #   Plot difference in two orbit proagation methods
81  t = t/86400
82  fig = plt.figure(2)
83  ax = fig.gca()
84  plt.plot(t, diff)
85  ax.set_xlabel('time [day]')
86  ax.set_ylabel('Difference in position between Lagrange/
    Gibbs and Poliastro')
87  plt.show()
88
89
```

```python
from src.dto import PropParams
from src.state_propagator import state_propagate
from src.interface.tle_dto import TLE

tle_string = """
STARLINK-1466
1 45732U 20038C   20192.83334491 -.01176717  00000-0 -28545
-1 0  9990
2 45732  53.0016 162.9981 0001205  64.1534 251.7734 15.
43303367  5600
"""

tle = TLE.from_lines(tle_string)
print("Original")
print(tle.to_string())

x, epoch = tle.to_state()
params = PropParams(epoch)
epoch_new = epoch + tle.period/2
x_new = state_propagate(x, epoch_new, params)
tle.update(x_new, epoch_new)
print("Updated")
print(tle.to_string())


```

```
 1 import numpy as np
 2 import scipy.linalg as la
 3 from astropy.time import Time
 4 from astropy.coordinates import solar_system_ephemeris,
   get_body_barycentric
 5 import astropy.units as u
 6 from verification.util import build_observations,
   build_epochs, get_satellite_position_over_time
 7 from src.interface.tle_dto import TLE
 8 from src.enums import Frames
 9 from src.dto import PropParams
10 from src.interface.local_angles import
   get_local_angles_via_state_propagation
11 from src.frames import eci_to_icrs, lla_to_ecef,
   ecef_to_eci
12 from src.observation_function import get_ra_and_dec
13
14
15 tle_string = """
16 2020-040A
17 1 45807U 20040A   20175.61197145 -.00000151  00000-0 -10000
   -3 0  9992
18 2 45807   0.0000 294.5058 0000000 179.9989 70.3936 2.
   28021460     116
19 """
20 obs_pos = [29.218103 * u.deg, -81.031723 * u.deg, 0 * u.km]
21 tle = TLE.from_lines(tle_string)
22
23 x, epoch = tle.to_state()
24 params = PropParams(epoch)
25 desired_epoch = Time("2020-06-24T00:00:00.000", format="
   isot", scale="tdb")
26 dt = 1
27 epochs = build_epochs(desired_epoch, dt * u.h, 24)
28
29 # obj_eci, epochs = get_satellite_position_over_time(x,
   epochs)
30 # for i in range(len(epochs)):
31     # obj = eci_to_icrs(positions[i], epochs[i])
32     # obs = eci_to_icrs(ecef_to_eci(lla_to_ecef(obs_pos),
   epochs[1]), epochs[i])
33     # print(get_ra_and_dec(rr))
34     # print(rr)
35     # print(positions[i])
36     # print("norms")
37     # print(la.norm(positions[i]))
```

```
38
39 # observations = build_observations(x, params, obs_pos,
   Frames.LLA, epochs)
40
41 # for obs in observations:
42 #     print(obs.obs_values)
43 #
44 n = len(epochs)
45 locals = get_local_angles_via_state_propagation(x, params,
   epochs[0], epochs[n-1], n-2, obs_pos, Frames.LLA)
46 for local in locals:
47     local[2].format = 'isot'
48     print(local)
49
50
51  # Date__(UT)__HR:MN     R.A.___(ICRF)___DEC R.A._(a-appar)
   _DEC. Azi_(a-appr)_Elev
52  # 2020-Jul-16 08:00  m   71.39000  17.78207  71.67881  17.
   81745  70.5231   1.7878
53  # 2020-Jul-16 09:00  m   71.41659  17.78600  71.70544  17.
   82133  77.3749  14.3731
54  # 2020-Jul-16 10:00 Nm   71.44291  17.78992  71.73179  17.
   82521  83.9301  27.2924
55  # 2020-Jul-16 11:00 *m   71.46897  17.79381  71.75789  17.
   82906  90.9000  40.3707
56  # 2020-Jul-16 12:00 *m   71.49483  17.79766  71.78378  17.
   83286  99.5451  53.4093
57  # 2020-Jul-16 13:00 *m   71.52053  17.80144  71.80951  17.
   83660 113.2688  65.9919
58  # 2020-Jul-16 14:00 *m   71.54614  17.80514  71.83514  17.
   84026 145.2445  76.3856
59  # 2020-Jul-16 15:00 *m   71.57175  17.80876  71.86077  17.
   84384 210.4835  76.9496
60  # 2020-Jul-16 16:00 *m   71.59743  17.81229  71.88646  17.
   84733 245.1981  66.9476
61  # 2020-Jul-16 17:00 *m   71.62327  17.81575  71.91230  17.
   85074 259.6529  54.4415
62  # 2020-Jul-16 18:00 *m   71.64932  17.81914  71.93836  17.
   85409 268.5288  41.4208
63  # 2020-Jul-16 19:00 *m   71.67566  17.82248  71.96469  17.
   85739 275.5757  28.3407
64  # 2020-Jul-16 20:00 *m   71.70232  17.82579  71.99134  17.
   86065 282.1376  15.4071
65  # 2020-Jul-16 21:00 *m   71.72932  17.82910  72.01834  17.
   86391 288.9475   2.7941
```

```python
from astropy.time import Time
import astropy.units as u
from src.enums import Frames
from src.dto import PropParams
from src.interface.local_angles import
get_local_angles_via_state_propagation
from src.interface.tle_dto import TLE


# x = [5748.5350, 2679.6404, 3442.8654, 4.328274, -1.918662
, -5.727629]
# epoch = Time(2449746.610150, format="jd", scale="utc")
# epoch.format = "isot"
# params = PropParams(epoch)
# epoch_i = Time("1995-01-29T02:38:37.000", format="isot",
scale="utc")
# epoch_f = Time("1995-01-29T02:40:27.000", format="isot",
scale="utc")
# obs_pos = [21.57 * u.deg, -158.27 * u.deg, .3002 * u.km]
#
# locals = get_local_angles_via_state_propagation(x, params
, epoch_i, epoch_f, 8, obs_pos, Frames.LLA)
# for local in locals:
#     print(local)


tle_string = """
Vallado
1 45732U 20038C   20192.83334491 -.01176717  00000-0 -28545
-1 0  9990
2 45732  53.0016 162.9981 0001205  64.1534 251.7734 15.
43303367  5600
"""
x = [5748.5350, 2679.6404, 3442.8654, 4.328274, -1.918662
, -5.727629]
epoch = Time(2449746.610150, format="jd", scale="utc")

tle = TLE.from_lines(tle_string)
tle.update(x, epoch)
tle.rev_num = 0
print(tle.to_string())
```

```
 1 import math
 2 from src.constants import mu
 3 import numpy as np
 4 from astropy.time import Time
 5 import astropy.units as u
 6 from verification.util import
   get_satellite_position_over_time
 7 from src.frames import eci_to_ecef, ecef_to_lla,
   ecef_to_eci
 8 import matplotlib.pyplot as plt
 9
10 # epoch = Time("2000-01-01T00:00:00.000", format="isot",
   scale="utc")
11 # epoch.format = "jd"
12 day_val = 1/365.25
13 epoch = Time(1984, format='decimalyear', scale="utc")
14 period = 86164.1                        # Seconds in a
   sidereal day
15 dt = period/100
16 tf = period
17 t = np.arange(0, tf, dt)
18 n = t.shape[0]
19 a = math.pow(math.pow(period/2/math.pi, 2) * mu.value, 1/3)
20 speed = math.sqrt(mu.value/a)
21 r_0 = np.array([a, 0, 0])
22 x = np.array([a, 0, 0, 0, speed, 0])
23 # norm = np.linalg.norm(x)
24 # r_0eci = ecef_to_eci(r_0, epoch)
25 # v_0eci = np.cross(r_0eci, np.array([0, 0, speed])) / np.
   linalg.norm(r_0eci)
26 # x = np.concatenate([r_0eci, v_0eci])
27 # nomr2 = np.linalg.norm(x)
28 # x = np.array([r_0eci[0], r_0eci[1], r_0eci[2], v_0_eci[0
   ], v_0_eci[1], v_0_eci[2]])
29 r_eci = get_satellite_position_over_time(x, epoch, tf, dt)
30
31 r_ecef = np.zeros((n, 3))
32 r_lla = np.zeros((n, 3))
33 for i in range(0, n):
34     time = epoch + t[i] * u.s
35     r_ecef[i] = eci_to_ecef(r_eci[i], time)
36     temp = ecef_to_lla(r_ecef[i])
37     r_lla[i] = np.array([temp[0].value, temp[1].value, temp
   [2]. value])
38
39 fig = plt.figure(1)
```

```
40 ax = fig.gca()
41 plt.plot(r_lla[:, 1], r_lla[:, 0], 'o')
42 ax.set_ylabel('Latitude [deg]')
43 ax.set_xlabel('Longitude [deg]')
44 plt.show()
45
46 # fig = plt.figure(2)
47 # ax = fig.gca()
48 # plt.plot(t, r_lla[:, 2])
49 # ax.set_xlabel('time [s]')
50 # ax.set_ylabel('Altitude [km]')
51 # plt.show()
52
53 print(r_lla[:, 1])
```

```python
import numpy as np
from verification.util import build_noisy_observations, build_epochs
from src.core import milani
from src.dto import PropParams
from src.enums import Frames
from astropy.time import Time
import astropy.units as u
from verification.util import get_period

r = [-27828.9136, -31685.0220, 3.5110]
v = [2.3098, -2.0286, -.0019]
x = np.array([r[0], r[1], r[2], v[0], v[1], v[2]])
x_offset = np.array([5000, 1000, 1000, .2, .2, .01])
x_true = x + x_offset
period = get_period(x)
dt = period / 100
tf = period * 2
epoch = Time(2449746.610150, format="jd", scale="utc")

obs_pos = [21.57 * u.deg, -158.27 * u.deg, .3002 * u.km] # Kaena Point, HI
prop_params = PropParams(epoch)
step = period/32 * u.s
epochs = build_epochs(epoch, step, 10)
observations = build_noisy_observations(x_true, prop_params, obs_pos, Frames.LLA, epochs, noise=5/60)
output = milani(x, observations, prop_params)
x_alg = output.x_out
p = output.p


print("State residual")
print(x_true - x_alg)
print("Uncertainty")
for val in np.diag(p):
    print(np.sqrt(val))
```

```python
import numpy as np
from verification.util import build_noisy_observations, build_epochs
from src.core import milani
from src.dto import PropParams
from src.enums import Frames
from astropy.time import Time
import astropy.units as u
from verification.util import get_period

r = [5748, 2679, 3443]
v = [4.33, -1.922, -5.726]
x = np.array([r[0], r[1], r[2], v[0], v[1], v[2]])
x_offset = np.array([500, 100, 100, .2, .1, .1])
x_true = x + x_offset
period = get_period(x)
dt = period / 100
tf = period * 2
epoch = Time(2449746.610150, format="jd", scale="utc")

obs_pos = [21.57 * u.deg, -158.27 * u.deg, .3002 * u.km] # Kaena Point, HI
prop_params = PropParams(epoch)
step = period/32 * u.s
epochs = build_epochs(epoch, step, 10)
observations = build_noisy_observations(x_true, prop_params, obs_pos, Frames.LLA, epochs, noise=5/60)
output = milani(x, observations, prop_params)
x_alg = output.x_out
p = output.p

print("State residual")
print(x_true - x_alg)
print("Uncertainty")
for val in np.diag(p):
    print(np.sqrt(val))

```

```python
import numpy as np
from verification.util import build_noisy_observations, build_epochs
from src.core import milani
from src.dto import PropParams
from src.enums import Frames
from astropy.time import Time
import astropy.units as u
from verification.util import get_period

r = [32000, 0, 0]
v = [0, 2, 0]
x = np.array([r[0], r[1], r[2], v[0], v[1], v[2]])
x_offset = np.array([5000, 1000, 1000, .2, .2, .01])
x_true = x + x_offset
period = get_period(x)
dt = period / 100
tf = period * 2
epoch = Time(2449746.610150, format="jd", scale="utc")

obs_pos = [21.57 * u.deg, -158.27 * u.deg, .3002 * u.km] # Kaena Point, HI
prop_params = PropParams(epoch)
step = period/32 * u.s
epochs = build_epochs(epoch, step, 10)
observations = build_noisy_observations(x_true, prop_params, obs_pos, Frames.LLA, epochs, noise=5/60)
output = milani(x, observations, prop_params)
x_alg = output.x_out
p = output.p


print("State residual")
print(x_true - x_alg)
print("Uncertainty")
for val in np.diag(p):
    print(np.sqrt(val))
```

```python
import numpy as np
from verification.util import build_noisy_observations, build_epochs
from src.core import milani
from src.dto import PropParams
from src.enums import Frames
from astropy.time import Time
import astropy.units as u
from verification.util import get_period


r = [6600, 0, 0]
v = [0, 10, 0]
x = np.array([r[0], r[1], r[2], v[0], v[1], v[2]])
x_offset = np.array([500, 100, 100, .2, .2, .1])
x_true = x + x_offset
period = get_period(x)
dt = period / 100
tf = period * 2
epoch = Time(2449746.610150, format="jd", scale="utc")

obs_pos = [21.57 * u.deg, -158.27 * u.deg, .3002 * u.km] # Kaena Point, HI
prop_params = PropParams(epoch)
step = period/32 * u.s
epochs = build_epochs(epoch, step, 10)
observations = build_noisy_observations(x_true, prop_params, obs_pos, Frames.LLA, epochs, noise=5/60)
output = milani(x, observations, prop_params)
x_alg = output.x_out
p = output.p


print("State residual")
print(x_true - x_alg)
print("Uncertainty")
for val in np.diag(p):
    print(np.sqrt(val))
```

9.3 Tests

```python
import mockito
import numpy as np


def xcompare(a, b):
    if isinstance(a, mockito.matchers.Matcher):
        return a.matches(b)
    return np.array_equal(a, b)

```

```
 1 from src import core
 2 from src.core import *
 3 import mockito
 4 from mockito import when, patch
 5 import pytest
 6 import numpy as np
 7 from test import xcompare
 8 from astropy.time import Time
 9 import astropy.units as u
10
11
12 def test_direction_isolator():
13     delta = np.array([1, 2, 3, 4, 5, 6])
14     j = 2
15     experimental = direction_isolator(delta, j)
16     theoretical = np.array([0, 0, 3, 0, 0, 0])
17     assert np.array_equal(theoretical, experimental)
18
19
20 def test_derivative():
21     x = np.array([10, 20, 30, 40, 50, 60])
22     delta = np.array([1, 2, 3, 4, 5, 6])
23     dt = 1
24     epoch_obs = None
25     observation = Observation(None, None, epoch_obs, None,
   None)
26     params = None
27     with patch(mockito.invocation.MatchingInvocation.
   compare, xcompare):
28         when(core).state_propagate(np.array([11, 20, 30, 40
   , 50, 60]), epoch_obs, params).thenReturn(np.array([11, 20
   , 30, 40, 50, 60]))
29         when(core).state_propagate(np.array([9, 20, 30, 40
   , 50, 60]), epoch_obs, params).thenReturn(np.array([9, 20,
   30, 40, 50, 60]))
30         when(core).state_propagate(np.array([10, 22, 30, 40
   , 50, 60]), epoch_obs, params).thenReturn(np.array([10, 22
   , 30, 40, 50, 60]))
31         when(core).state_propagate(np.array([10, 18, 30, 40
   , 50, 60]), epoch_obs, params).thenReturn(np.array([10, 18
   , 30, 40, 50, 60]))
32         when(core).state_propagate(np.array([10, 20, 33, 40
   , 50, 60]), epoch_obs, params).thenReturn(np.array([10, 20
   , 33, 40, 50, 60]))
33         when(core).state_propagate(np.array([10, 20, 27, 40
   , 50, 60]), epoch_obs, params).thenReturn(np.array([10, 20
```

```
33 , 27, 40, 50, 60]))
34         when(core).state_propagate(np.array([10, 20, 30, 44
   , 50, 60]), epoch_obs, params).thenReturn(np.array([10, 20
   , 30, 44, 50, 60]))
35         when(core).state_propagate(np.array([10, 20, 30, 36
   , 50, 60]), epoch_obs, params).thenReturn(np.array([10, 20
   , 30, 36, 50, 60]))
36         when(core).state_propagate(np.array([10, 20, 30, 40
   , 55, 60]), epoch_obs, params).thenReturn(np.array([10, 20
   , 30, 40, 55, 60]))
37         when(core).state_propagate(np.array([10, 20, 30, 40
   , 45, 60]), epoch_obs, params).thenReturn(np.array([10, 20
   , 30, 40, 45, 60]))
38         when(core).state_propagate(np.array([10, 20, 30, 40
   , 50, 66]), epoch_obs, params).thenReturn(np.array([10, 20
   , 30, 40, 50, 66]))
39         when(core).state_propagate(np.array([10, 20, 30, 40
   , 50, 54]), epoch_obs, params).thenReturn(np.array([10, 20
   , 30, 40, 50, 54]))
40         when(core).y(np.array([11, 20, 30, 40, 50, 60]),
   observation).thenReturn(np.array([2, 4]))
41         when(core).y(np.array([9, 20, 30, 40, 50, 60]),
   observation).thenReturn(np.array([0, 0]))
42         when(core).y(np.array([10, 22, 30, 40, 50, 60]),
   observation).thenReturn(np.array([12, 16]))
43         when(core).y(np.array([10, 18, 30, 40, 50, 60]),
   observation).thenReturn(np.array([0, 0]))
44         when(core).y(np.array([10, 20, 33, 40, 50, 60]),
   observation).thenReturn(np.array([30, 36]))
45         when(core).y(np.array([10, 20, 27, 40, 50, 60]),
   observation).thenReturn(np.array([0, 0]))
46         when(core).y(np.array([10, 20, 30, 44, 50, 60]),
   observation).thenReturn(np.array([56, 64]))
47         when(core).y(np.array([10, 20, 30, 36, 50, 60]),
   observation).thenReturn(np.array([0, 0]))
48         when(core).y(np.array([10, 20, 30, 40, 55, 60]),
   observation).thenReturn(np.array([90, 100]))
49         when(core).y(np.array([10, 20, 30, 40, 45, 60]),
   observation).thenReturn(np.array([0, 0]))
50         when(core).y(np.array([10, 20, 30, 40, 50, 66]),
   observation).thenReturn(np.array([132, 144]))
51         when(core).y(np.array([10, 20, 30, 40, 50, 54]),
   observation).thenReturn(np.array([0, 0]))
52         experimental = dy_dstate(x, delta, observation,
   params)
53         theoretical = np.array(([1, 3, 5, 7, 9, 11], [2, 4
```

```
53 , 6, 8, 10, 12]))
54         assert np.array_equal(theoretical, experimental)
55
56
57 @pytest.mark.parametrize("rms_new, rms_old, tol, expected"
   , [(0, 10, 1, True), (10, 1, 1, False), (1, 10, 1, True)])
58 def test_stopping_criteria(rms_new, rms_old, tol, expected
   ):
59     result = stopping_criteria(rms_new, rms_old, tol=tol)
60     assert result == expected
61
62
63 def test_diagonal_form():
64     a = np.array([[5, 2, -1, 0, 0],
65                   [1, 4, 2, -1, 0],
66                   [0, 1, 3, 2, -1],
67                   [0, 0, 1, 2, 2],
68                   [0, 0, 0, 1, 1]])
69     b = np.array([0, 1, 2, 2, 3])
70     ab = diagonal_form(a, upper=2, lower=1)
71     expected = np.array([[0, 0, -1, -1, -1],
72                          [0, 2, 2, 2, 2],
73                          [5, 4, 3, 2, 1],
74                          [1, 1, 1, 1, 0]])
75     assert np.allclose(ab, expected)
76
77
78 def test_get_delta_x():
79     a = np.array([[5, 2, -1, 0, 0],
80                   [1, 4, 2, -1, 0],
81                   [0, 1, 3, 2, -1],
82                   [0, 0, 1, 2, 2],
83                   [0, 0, 0, 1, 1]])
84     b = np.array([0, 1, 2, 2, 3])
85     ab = diagonal_form(a, upper=2, lower=1)
86     x = solve_banded((1, 2), ab, b)
87     residual = a @ x - b
88     assert np.allclose(residual, np.zeros((6, 1)))
89
90
91 def test_milani():
92     epoch = Time(2454283.0, format="jd", scale="tdb")
93     epoch_obs = epoch + 1 * u.day
94     obs_val = np.array([1, 1])
95     obs = Observation(None, None, epoch_obs, obs_val, None)
96     x = np.array([1, 2, 3, 4, 5, 6])
```

```
 97     params = PropParams(epoch)
 98     dr = 1
 99     dv = 2
100     b = np.ones((2, 6))
101 
102     expected = FilterOutput(x, params.epoch, x, np.zeros(6
    ), np.eye(6))
103     with patch(mockito.invocation.MatchingInvocation.
    compare, xcompare):
104         when(core).state_propagate(x, epoch_obs, params).
    thenReturn(np.ones(6))
105         when(core).y(np.ones(6), obs).thenReturn(np.zeros(
    2))
106         when(core).dy_dstate(x, np. array([dr, dr, dr, dv
    , dv, dv]), obs, params).thenReturn(b)
107         when(core).get_delta_x(b.T @ b, b.T @ obs_val).
    thenReturn(np.zeros(6))
108         when(core).stopping_criteria(1e8, 1e10).thenReturn
    (False)
109         when(core).stopping_criteria(np.sqrt(obs_val.T @
    obs_val/6), 1e8).thenReturn(True)
110         when(core).get_inverse(b.T @ b).thenReturn(np.eye(
    6))
111         actual = milani(x, [obs], params, dr=dr, dv=dv)
112 
113     assert actual.epoch == expected.epoch
114     assert np.array_equal(actual.x_in, expected.x_in)
115     assert np.array_equal(actual.x_out, expected.x_out)
116     assert np.array_equal(actual.delta_x, expected.delta_x
    )
117     assert np.array_equal(actual.p, expected.p)
118 
119 
120 
```

```python
import astropy.units as u
from src.perturbation_util import *
import numpy as np
from src.frames import lla_to_ecef, ecef_to_lla, ecef_to_eci, eci_to_ecef
import pytest


@pytest.mark.parametrize("input, expected", [([90 * u.deg, 0 * u.deg, 0 * u.km], np.array([0, 0, 6356.75231])),
                                             ([0 * u.deg, 0 * u.deg, 0 * u.km], np.array([6378.137, 0, 0]))])
def test_lla_to_ecef(input, expected):
    actual = lla_to_ecef(input)
    assert np.allclose(actual, expected)


@pytest.mark.parametrize("input, expected", [(np.array([0, 0, 6356.75231]), [90 * u.deg, 0 * u.deg, -4.2451791e-6 * u.km]),
                                             (np.array([6378.137, 0, 0]), [0 * u.deg, 0 * u.deg, 0 * u.km])])
def test_ecef_to_lla(input, expected):
    actual = ecef_to_lla(input)
    for i in range(3):
        assert expected[i].unit == actual[i].unit
        thing = np.array(expected[i].value)
        thong = np.array(actual[i].value)
        assert np.allclose(thing, thong)


@pytest.mark.parametrize("input", [(np.array([0, 0, 6356.75231])),
                                   (np.array([6378.137, 0, 0]))])
def test_eci_to_ecef_and_back(input):
    epoch = Time("2018-08-17 12:05:50", scale="tdb")
    middle = eci_to_ecef(input, epoch)
    actual = ecef_to_eci(middle, epoch)
    assert np.linalg.norm(actual - input) < 1e-7
#   To supplement the there and back. Ground tracks will be observed in a verification file.

```

```
 1 import numpy as np
 2 from src import observation_function
 3 from src.observation_function import *
 4 from src.enums import Frames
 5 from src.dto import Observation
 6 import mockito
 7 import pytest
 8 import mockito
 9 from mockito import patch, when
10 from test import xcompare
11
12
13 @pytest.mark.parametrize("rr, expected", [(np.array([100,
   100, 0]), np.array([45, 0])),
14                                           (np.array([100, 0
   , 0]), np.array([0, 0])),
15                                           (np.array([0, 100
   , 0]), np.array([90, 0])),
16                                           (np.array([0, 0
   , -100]), np.array([0, -90]))])
17 def test_get_ra_and_dec(rr, expected):
18     actual = get_ra_and_dec(rr)
19     assert np.array_equal(actual, expected)
20
21
22 def test_y_with_eci_frame():
23     position = np.array([100, 100, 100])
24     epoch = None
25     obs_values = None
26     obs_type = None
27
28     obs_params = Observation(position, Frames.ECI, epoch,
   obs_values, obs_type)
29     x = np.array([100, 200, 100])
30     expected = np.array([90, 0])
31     actual = y(x, obs_params)
32     assert np.array_equal(expected, actual)
33
```

```python
import numpy as np
from src.state_propagator import a_d, state_propagate
from src.perturbation_util import build_j2, build_j3, build_basic_drag, build_lunar_third_body, build_solar_third_body, build_srp
from poliastro.ephem import build_ephem_interpolant
from poliastro.bodies import Moon, Sun
from astropy.coordinates import solar_system_ephemeris
from src.enums import Perturbations
from src.constants import lunar_period, solar_period
from poliastro.twobody import Orbit
from poliastro.twobody.propagation import cowell
from poliastro.bodies import Earth
from poliastro.core.perturbations import J2_perturbation, J3_perturbation, atmospheric_drag_exponential, third_body, \
    radiation_pressure
from poliastro.constants import H0_earth, rho0_earth, Wdivc_sun
from astropy import units as u
from astropy.time import Time
from src.dto import PropParams


solar_system_ephemeris.set("de432s")
R = Earth.R.to(u.km).value


def test_propagate_with_j2j3():
    x = [66666, 0, 0, 0, 2.451, 0]
    dt = 100 * u.day
    r = x[0:3] * u.km
    v = x[3:6] * u.km / u.s
    epoch = Time(2454283.0, format="jd", scale="tdb")
    epoch_obs = epoch + dt

    prop_params = PropParams(epoch)
    prop_params.add_perturbation(Perturbations.J2, build_j2())
    prop_params.add_perturbation(Perturbations.J3, build_j3())

    sat_i = Orbit.from_vectors(Earth, r, v, epoch=epoch)
    sat_f = sat_i.propagate(dt, method=cowell, ad=a_d_j2j3, J2=Earth.J2.value, R=R, J3=Earth.J3.value)
    x_poli = np.concatenate([sat_f.r.value, sat_f.v.value])
```

```
39
40      x_custom = state_propagate(x, epoch_obs, prop_params)
41      assert np.array_equal(x_custom, x_poli)
42
43
44  def test_propagate_with_drag():
45      x = [66666, 0, 0, 0, 2.451, 0]
46      dt = 100 * u.s
47      r = x[0:3] * u.km
48      v = x[3:6] * u.km / u.s
49      epoch = Time(2454283.0, format="jd", scale="tdb")
50      epoch_obs = epoch + dt
51      C_D = 1
52      A = 10
53      m = 1000
54      Drag = build_basic_drag(C_D, A, m)
55      prop_params = PropParams(epoch)
56      prop_params.add_perturbation(Perturbations.Drag, Drag)
57
58      sat_i = Orbit.from_vectors(Earth, r, v, epoch=epoch)
59      sat_f = sat_i.propagate(dt, method=cowell, ad=
    atmospheric_drag_exponential, R=R, C_D=C_D, A_over_m=A/m,
    H0=H0_earth,
60                              rho0=rho0_earth)
61      x_poli = np.concatenate([sat_f.r.value, sat_f.v.value])
62
63      x_custom = state_propagate(x, epoch_obs, prop_params)
64      assert np.array_equal(x_custom, x_poli)
65
66
67  def test_propagate_with_no_perturbations():
68      x = [66666, 0, 0, 0, 2.451, 0]
69      r = x[0:3] * u.km
70      v = x[3:6] * u.km / u.s
71      dt = 100 * u.day
72      epoch = Time(2454283.0, format="jd", scale="tdb")
73      epoch_obs = epoch + dt
74      prop_params = PropParams(epoch)
75
76      sat_i = Orbit.from_vectors(Earth, r, v, epoch=epoch)
77      sat_f = sat_i.propagate(dt, method=cowell)
78      x_poli = np.concatenate([sat_f.r.value, sat_f.v.value])
79
80      x_custom = state_propagate(x, epoch_obs, prop_params)
81      assert np.array_equal(x_custom, x_poli)
82
```

```
 83
 84 def test_propagate_with_lunar_third_body():
 85     x = [66666, 0, 0, 0, 2.451, 0]
 86     dt = 8600
 87     r = x[0:3] * u.km
 88     v = x[3:6] * u.km / u.s
 89     epoch = Time(2454283.0, format="jd", scale="tdb")
 90     epoch_f = epoch + (dt * u.s)
 91     prop_params = PropParams(epoch)
 92     prop_params.add_perturbation(Perturbations.Moon,
    build_lunar_third_body(epoch))
 93
 94     k_moon = Moon.k.to(u.km ** 3 / u.s ** 2).value
 95     body_moon = build_ephem_interpolant(Moon, lunar_period
    , (epoch.value * u.day,
 96
       epoch.value * u.day + 60 * u.day), rtol=1e-2)
 97     sat_i = Orbit.from_vectors(Earth, r, v, epoch=epoch)
 98     sat_f = sat_i.propagate(dt * u.s, method=cowell, ad=
    third_body, k_third=k_moon, perturbation_body=body_moon)
 99     x_poli = np.concatenate([sat_f.r.value, sat_f.v.value
    ])
100
101     x_custom = state_propagate(np.array(x), epoch_f,
    prop_params)
102     assert np.array_equal(x_custom, x_poli)
103
104
105 def test_propagate_with_solar_third_body():
106     x = [66666, 0, 0, 0, 2.451, 0]
107     dt = 1 * u.day
108     r = x[0:3] * u.km
109     v = x[3:6] * u.km / u.s
110     epoch = Time(2454283.0, format="jd", scale="tdb")
111     epoch_obs = epoch + dt
112     prop_params = PropParams(epoch)
113     prop_params.add_perturbation(Perturbations.Sun,
    build_solar_third_body(epoch))
114
115     k_sun = Sun.k.to(u.km ** 3 / u.s ** 2).value
116     body_sun = build_ephem_interpolant(Sun, solar_period
    , (epoch.value * u.day,
117
     epoch.value * u.day + 60 * u.day), rtol=1e-2)
118     sat_i = Orbit.from_vectors(Earth, r, v, epoch=epoch)
119     sat_f = sat_i.propagate(dt, method=cowell, ad=
```

```
119 third_body, k_third=k_sun, perturbation_body=body_sun)
120     x_poli = np.concatenate([sat_f.r.value, sat_f.v.value
    ])
121
122     x_custom = state_propagate(x, epoch_obs, prop_params)
123     assert np.array_equal(x_custom, x_poli)
124
125
126 def test_propagate_with_srp():
127     x = [66666, 0, 0, 0, 2.451, 0]
128     dt = 1 * u.day
129     r = x[0:3] * u.km
130     v = x[3:6] * u.km / u.s
131     epoch = Time(2454283.0, format="jd", scale="tdb")
132     epoch_obs = epoch + dt
133     C_R = 1
134     A = 10
135     m = 1000
136     srp = build_srp(C_R, A, m, epoch)
137     prop_params = PropParams(epoch)
138     prop_params.add_perturbation(Perturbations.SRP, srp)
139
140     body_sun = build_ephem_interpolant(Sun, solar_period
    , (epoch.value * u.day,
141
     epoch.value * u.day + 60 * u.day), rtol=1e-2)
142
143     sat_i = Orbit.from_vectors(Earth, r, v, epoch=epoch)
144     sat_f = sat_i.propagate(dt, method=cowell, ad=
    radiation_pressure,
145                             R=R, C_R=C_R, A_over_m=A/m,
    Wdivc_s=Wdivc_sun.value, star=body_sun)
146     x_poli = np.concatenate([sat_f.r.value, sat_f.v.value
    ])
147
148     x_custom = state_propagate(x, epoch_obs, prop_params)
149     assert np.array_equal(x_custom, x_poli)
150
151
152 def a_d_j2j3(t0, state, k, J2, J3, R):
153     return J2_perturbation(t0, state, k, J2, R) +
    J3_perturbation(t0, state, k, J3, R)
154
155
```

```
 1 import numpy as np
 2 import mockito
 3 from mockito import when, patch
 4 from test import xcompare
 5 from src import covariance_propagator
 6 from src.covariance_propagator import *
 7
 8
 9 def test_cov_propagate():
10     x = np.array([1, 1, 1, 1, 1, 1])
11     epoch_t = None
12     prop_params = None
13     dr = .1
14     dv = .005
15     delta = np.array([dr, dr, dr, dv, dv, dv])
16     p_i = np.ones((6, 6))
17
18     with patch(mockito.invocation.MatchingInvocation.
   compare, xcompare):
19         when(covariance_propagator).dx_dx0(x, epoch_t,
   prop_params, delta).thenReturn(np.eye(6))
20         p_t = cov_propagate(x, epoch_t, prop_params, p_i)
21     assert np.array_equal(p_i, p_t)
22
23
24 def test_dx_dx0():
25     x = np.array([10, 20, 30, 40, 50, 60])
26     delta = np.array([1, 2, 3, 4, 5, 6])
27     epoch_t = None
28     params = None
29
30     expected = np.array([[1, 2, 3, 4, 5, 6],
31                          [7, 8, 9, 10, 11, 12],
32                          [13, 14, 15, 16, 17, 18],
33                          [19, 20, 21, 22, 23, 24],
34                          [25, 26, 27, 28, 29, 30],
35                          [31, 32, 33, 34, 35, 36]])
36
37     with patch(mockito.invocation.MatchingInvocation.
   compare, xcompare):
38         when(covariance_propagator).state_propagate(np.
   array([11, 20, 30, 40, 50, 60]), epoch_t, params).
   thenReturn(
39             np.array([2, 28, 78, 152, 250, 372]))
40         when(covariance_propagator).state_propagate(np.
   array([9, 20, 30, 40, 50, 60]), epoch_t, params).thenReturn
```

```
40 (
41             np.array([0, 0, 0, 0, 0, 0]))
42         when(covariance_propagator).state_propagate(np.
   array([10, 22, 30, 40, 50, 60]), epoch_t, params).
   thenReturn(
43             np.array([4, 32, 84, 160, 260, 384]))
44         when(covariance_propagator).state_propagate(np.
   array([10, 18, 30, 40, 50, 60]), epoch_t, params).
   thenReturn(
45             np.array([0, 0, 0, 0, 0, 0]))
46         when(covariance_propagator).state_propagate(np.
   array([10, 20, 33, 40, 50, 60]), epoch_t, params).
   thenReturn(
47             np.array([6, 36, 90, 168, 270, 396]))
48         when(covariance_propagator).state_propagate(np.
   array([10, 20, 27, 40, 50, 60]), epoch_t, params).
   thenReturn(
49             np.array([0, 0, 0, 0, 0, 0]))
50         when(covariance_propagator).state_propagate(np.
   array([10, 20, 30, 44, 50, 60]), epoch_t, params).
   thenReturn(
51             np.array([8, 40, 96, 176, 280, 408]))
52         when(covariance_propagator).state_propagate(np.
   array([10, 20, 30, 36, 50, 60]), epoch_t, params).
   thenReturn(
53             np.array([0, 0, 0, 0, 0, 0]))
54         when(covariance_propagator).state_propagate(np.
   array([10, 20, 30, 40, 55, 60]), epoch_t, params).
   thenReturn(
55             np.array([10, 44, 102, 184, 290, 420]))
56         when(covariance_propagator).state_propagate(np.
   array([10, 20, 30, 40, 45, 60]), epoch_t, params).
   thenReturn(
57             np.array([0, 0, 0, 0, 0, 0]))
58         when(covariance_propagator).state_propagate(np.
   array([10, 20, 30, 40, 50, 66]), epoch_t, params).
   thenReturn(
59             np.array([12, 48, 108, 192, 300, 432]))
60         when(covariance_propagator).state_propagate(np.
   array([10, 20, 30, 40, 50, 54]), epoch_t, params).
   thenReturn(
61             np.array([0, 0, 0, 0, 0, 0]))
62         result = dx_dx0(x, epoch_t, params, delta)
63     assert np.array_equal(expected,  result)
64
```

```python
from src.interface.tle_dto import TLE
import numpy as np
from src.dto import PropParams
from astropy.time import Time


def test_tle_to_state():
    tle_string = """
ISS (ZARYA)
1 25544U 98067A   20171.48973192 -.00009734  00000-0 -16612-3 0  9998
2 25544  51.6439 337.0700 0002381  67.4024  37.3218 15.49440074232376
"""
    tle = TLE.from_lines(tle_string)
    sat = tle.to_orbit()
    x_expected = np.concatenate([sat.r.value, sat.v.value])

    epoch_yr = 2020
    epoch_day = 171.48973192
    epoch_expected = Time(epoch_yr + epoch_day / 365.25, format="decimalyear", scale="utc")

    x_actual, epoch_out = tle.to_state()
    assert np.array_equal(x_expected, x_actual)
    assert epoch_out == epoch_expected


# def test_tle_to_string_loop():
#     tle_in = """
# ISS (ZARYA)
# 1 25544U 98067A   20171.48973192 -.00009734  00000-0 -16612-3 0  9998
# 2 25544  51.6439 337.0700 0002381  67.4024  37.3218 15.49440074232376
# """
#     tle = TLE.from_lines(tle_in)
#     tle_out = tle.to_string()
#     assert tle_in == tle_out

```

```
 1 from src.dto import Observation
 2 from src.enums import Frames
 3 import mockito
 4 from test.test_core import xcompare
 5 from mockito import patch, when
 6 import astropy.units as u
 7 from src.interface.cleaning import
   convert_obs_from_lla_to_ecef, convert_obs_from_lla_to_eci
   , \
 8     convert_obs_from_ecef_to_eci, verify_locational_units
 9 from src.interface import cleaning
10 import numpy as np
11 import math
12
13
14 def test_convert_obs_params_from_lla_to_ecef():
15     input_pos = [0 * u.deg, 0 * u.deg, 0 * u.km]
16     output_pos = np.array([0, 0, 0])
17     epoch = None
18     obs_values = None
19     obs_type = None
20
21     input = Observation(input_pos, Frames.LLA, epoch,
   obs_values, obs_type)
22     with patch(mockito.invocation.MatchingInvocation.
   compare, xcompare):
23         when(cleaning).lla_to_ecef(input_pos).thenReturn(np
   .array(output_pos))
24     expected = Observation(output_pos, Frames.ECEF, epoch,
   obs_values, obs_type)
25     actual = convert_obs_from_lla_to_ecef(input)
26     assert np.array_equal(expected.position, actual.
   position)
27     assert expected.frame == actual.frame
28
29
30 def test_convert_obs_params_from_lla_to_eci():
31     input_pos = [0 * u.deg, 0 * u.deg, 0 * u.km]
32     output_pos = np.array([0, 0, 0])
33     epoch = None
34     obs_values = None
35     obs_type = None
36
37     input = Observation(input_pos, Frames.LLA, epoch,
   obs_values, obs_type)
38     with patch(mockito.invocation.MatchingInvocation.
```

```
38 compare, xcompare):
39         when(cleaning).lla_to_ecef(input_pos).thenReturn("
   Nothing")
40         when(cleaning).ecef_to_eci("Nothing", epoch).
   thenReturn(np.array(output_pos))
41     expected = Observation(output_pos, Frames.ECI, epoch,
   obs_values, obs_type)
42     actual = convert_obs_from_lla_to_eci(input)
43     assert np.array_equal(expected.position, actual.
   position)
44     assert expected.frame == actual.frame
45
46
47 def test_convert_obs_from_ecef_to_eci():
48     input_pos = [1 * u.km, 2 * u.km, 3 * u.km]
49     output_pos = np.array([0, 0, 0])
50     epoch = None
51     obs_values = None
52     obs_type = None
53
54     input = Observation(input_pos, Frames.ECEF, epoch,
   obs_values, obs_type)
55     with patch(mockito.invocation.MatchingInvocation.
   compare, xcompare):
56         when(cleaning).ecef_to_eci(input_pos, epoch).
   thenReturn(np.array(output_pos))
57     expected = Observation(output_pos, Frames.ECI, epoch,
   obs_values, obs_type)
58     actual = convert_obs_from_ecef_to_eci(input)
59     assert np.array_equal(expected.position, actual.
   position)
60     assert expected.frame == actual.frame
61
62
63 def test_verify_units_spacial():
64     position = [1000 * u.m, 1000 * u.m, 1000 * u.m]
65     epoch = None
66     obs_values = None
67     obs_type = None
68
69     obs_params_in = Observation(position, Frames.ECI, epoch
   , obs_values, obs_type)
70     expected_position = [1 * u.km, 1 * u.km, 1 * u.km]
71     expected_outcome = Observation(expected_position,
   Frames.ECI, epoch, obs_values, obs_type)
72     actual_outcome = verify_locational_units(obs_params_in)
```

```
73      assert expected_outcome.frame == actual_outcome.frame
74      for i in range(3):
75          assert expected_outcome.position[i] ==
   actual_outcome.position[i]
76
77
78  def test_verify_units_lla():
79      position = [math.pi*2 * u.rad, math.pi*2 * u.rad, 1000
     * u.m]
80      epoch = None
81      obs_values = None
82      obs_type = None
83
84      obs_params_in = Observation(position, Frames.LLA,
   epoch, obs_values, obs_type)
85      expected_position = [360 * u.deg, 360 * u.deg, 1 * u.
   km]
86      expected_outcome = Observation(expected_position,
   Frames.LLA, epoch, obs_values, obs_type)
87      actual_outcome = verify_locational_units(obs_params_in
   )
88      assert expected_outcome.frame == actual_outcome.frame
89      for i in range(3):
90          assert expected_outcome.position[i] ==
   actual_outcome.position[i]
91
```

```python
import numpy as np
import math
from src.interface.local_angles import *
import pytest
import astropy.units as u
from astropy.time import Time
from src.enums import Frames
from src.dto import PropParams
import mockito
from mockito import patch, when
from src.interface import local_angles as la
from test import xcompare

rt = math.sqrt(2)


@pytest.mark.parametrize("lla, expected", [([0 * u.deg, 0 * u.deg, 800 * u.km], np.array([-3, 2, 1])),
                                           ([0 * u.deg, 90 * u.deg, 800 * u.km], np.array([-3, -1, 2])),
                                           ([90 * u.deg, 0 * u.deg, 800 * u.km], np.array([1, 2, 3])),
                                           ([45 * u.deg, 45 * u.deg, 800 * u.km], np.array([(3-3*rt)/2, 1/rt, (3+3*rt)/2]))])
def test_r(lla, expected):
    rr = np.array([1, 2, 3])
    rot_mat = rotation_matrix(lla[0].value, lla[1].value)
    actual = rot_mat.T @ rr
    assert np.allclose(expected, actual)


norm = np.linalg.norm(np.array([1, 2, 3]))
rtd = 180 / np.pi


@pytest.mark.parametrize("lla, expected", [([90 * u.deg, 0 * u.deg, 0 * u.km],
                                            np.array([90 - np.arctan2(-1, 2) * rtd, np.arcsin(3/norm)*rtd])),
                                           ([0 * u.deg, 0 * u.deg, 0 * u.km],
                                            np.array([90 - np.arctan2(3, 2) * rtd, np.arcsin(1/norm)*rtd])),
                                           ([0 * u.deg, 90 * u.deg, 0 * u.km],
```

```
37                                                  np.array([90 -
   np.arctan2(3, -1) * rtd, np.arcsin(2/norm)*rtd])),
38                                                 ([-90 * u.deg, 0
    * u.deg, 0 * u.km],
39                                                  np.array([90 -
   np.arctan2(1, 2) * rtd, np.arcsin(-3/norm)*rtd]))])
40 def test_local_angles(lla, expected):
41     rr = np.array([1, 2, 3])
42     actual = local_angles(rr, lla)
43     assert np.allclose(expected, actual)
44
45 #
46 # def test_get_local_angles_for_state_prop():
47 #     x = np.array([1, 2, 3, 4, 5, 6])
48 #     obs_pos_lla = [1, 2, 3]
49 #     obs_frame = Frames.LLA
50 #     epoch_i = Time(2454283.0, format="jd", scale="tdb")
51 #     epoch_f = epoch_i + 1 * u.day
52 #     n = 0
53 #     params = PropParams(epoch_i)
54 #
55 #     mocked_angles1 = np.array([1, 2])
56 #     mocked_angles2 = np.array([2, 3])
57 #     mocked_state1 = np.array([7, 8, 9, 10, 11, 12])
58 #     mocked_state2 = np.array([13, 14, 15, 16, 17, 18])
59 #     expected = [[1, 2, epoch_i], [2, 3, epoch_f]]
60 #
61 #     obs_pos_ecef = np.array([4, 5, 6])
62 #
63 #     with patch(mockito.invocation.MatchingInvocation.
   compare, xcompare):
64 #         when(la).lla_to_ecef(obs_pos_lla).thenReturn(
   obs_pos_ecef)
65 #         when(la).ecef_to_eci(obs_pos_ecef, epoch_i).
   thenReturn(np.zeros(3))
66 #         when(la).ecef_to_eci(obs_pos_ecef, epoch_f).
   thenReturn(np.zeros(3))
67 #         when(la).state_propagate(x, epoch_i, params).
   thenReturn(mocked_state1)
68 #         when(la).state_propagate(x, epoch_f, params).
   thenReturn(mocked_state2)
69 #         when(la).local_angles(mocked_state1[0:3],
   obs_pos_lla).thenReturn(mocked_angles1)
70 #         when(la).local_angles(mocked_state2[0:3],
   obs_pos_lla).thenReturn(mocked_angles2)
71 #         actual = get_local_angles_via_state_propagation(x
```

```
71 , params, epoch_i, epoch_f, n, obs_pos_lla, obs_frame)
72 #     assert actual == expected
73
```

```
 1 from src.interface.string_conversions import dms_to_dd
 2 import pytest
 3
 4
 5 @pytest.mark.parametrize("input_string, expected", [("1 30
   ' 3600\"", 2.5),
 6                                                     ("0 0'
   0\"", 0)])
 7 def test_dms_to_dd(input_string, expected):
 8     actual = dms_to_dd(input_string)
 9     assert actual == expected
10
11
```